Java ile Nesneye Yönelik Programlama

Oğuz Aslantürk

Önsöz

Bu kitap en azından Yapısal Programlama bilen okuyucular için hazırlanmıştır. Temel programlama bilgileri anlatılmayacak, okuyucunun algoritma kavramını bildiği ve algoritma geliştirebildiği varsayılacaktır.

Amacımız Java programlama dilini ya da Java platformunu anlatmak değil, Nesneye Yönelik Programlama yaklaşımını açıklamak ve bu yaklaşım ile uygulama geliştirmeyi öğretmektir. Programlama konularını öğrenmenin en iyi yolunun programlar yazmak olduğunu düşündüğümüz için, nesneye yönelik programlar yazmak durumundayız ve bunun için bir dil kullanılması gerekiyor. Bu kitap için programlama dili Java olarak seçilmiştir ancak benzer içerik örneğin C++ dili ile de verilebilir.

Kitap içerisinde kullanılacak terimler için genelde Türkiye Bilişim Derneği sözlüğü esas alınacak, kullanılan Türkçe terimlerin İngilizce karşılıkları da parantez içinde verilmeye çalışılacaktır. Karşılığı henüz oturmamış ya da kullanıldığı bağlamda istenen anlamı veremediği düşünülen terimler için ise, yaygın kullanılmakta olan terimler tercih edilecek, Türkçe terimler ile karşılanamayan bazı sözcükler/teknolojiler için İngilizce terimler *italik* yazılacaktır.

Giriş

1 Nesneye Yönelik Programlamaya Giriş

Programlar, gerçek hayatta karşılaşılan bir takım problemlerin çözülmesi için bilgisayarların hızlı ve doğru işlem yapabilme yeteneklerinden faydalanmak üzere yazılırlar. Bilgisayarların hız ve kapasiteleri arttıkça geliştirilen programlar da bu hız ve kapasiteden faydalanabilecek şekilde gelişerek değişmektedir. Bu karşılıklı gelişme ve değişme, zaman içerisinde program geliştirme yöntemlerinde de değişikliklere neden olmuş, böylece çeşitli program geliştirme yaklaşımları ortaya çıkmıştır.

Nesneye Yönelik Yazılım Geliştirme (*Object Oriented Software Development*), bir yazılım geliştirme yaklaşımıdır. Nesneye yönelik yaklaşım dışında çeşitli yaklaşımlar da bulunmakla birlikte (Yapısal Programlama (*Structured Programming*), Bileşen Tabanlı Yazılım Geliştirme (*Component Based Software Development*), Bakışımlı Programlama (*Aspect Oriented Programming)...*) kendisinden önceki yaklaşımların bazı açıklarını kapatan, kendisinden sonraki yaklaşımların (çoğunun) da alt yapısını oluşturan bir yaklaşım olarak oldukça geniş kullanım alanı bulmuştur.

1.1 Modelleme

Gerçek hayattaki problemleri bilgisayarın sanal ortamında çözebilmek için, herşeyden önce problemin uygun şekilde bilgisayar ortamına aktarılması gerekmektedir. Bu işlem "soyutlama (*abstraction*)" ya da "modelleme (*modeling*)" olarak anılır.

Modelleme, insanın problem çözmek üzere eskiden beri kullandığı bir yöntemdir. Büyükçe bir problemin tamamını zihinde canlandırıp çözmeye çalışmak yerine, oluşturulacak model ya da modeller üzerinde hedef sistemin görünüşü, davranışı ya da bazı durumlarda verdiği tepkiler gözlemlenebilir.

Model, var olan ya da gerçekleştirilmesi planlanan bir sistemi anlamak ya da anlatmak üzere oluşturulabilir ve birçok farklı alanda etkili bir şekilde

kullanılmaktadır. Örneğin, bir toplu konut inşaatını müşterilerine tanıtmak isteyen bir inşaat firması, binaların yerleşimlerini, renk ve göreli büyüklüklerini görsel olarak ifade eden maket ya da maketler hazırlar. Bu maketi inceleyen bir kimse, almak istediği konutun nerede olduğunu, okul binasına yakınlığını ya da anayola ulaşımın nasıl olduğunu görerek değerlendirebilir. Burada model makettir ve "hedef sistemi anlatmak" amacını yerine getirmektedir.

Modelin mutlaka elle tutulur olması da gerekmez. Bilgisayar benzetimi ile de çeşitli modeller oluşturulabilir. Örneğin bir uçağın havadaki hareketini incelemek üzere geliştirilmiş bir bilgisayar benzetimi ile uçak modellenebilir. Kanat uzunluğu ya da gövde eğimi gibi parametrelerle oynanarak uçağın farklı hava koşullarında nasıl davranacağı anlaşılmaya çalışılabilir. Burada, sistemin davranışını anlamak amacıyla, sanal ortamda oluşturulmuş bir model söz konusudur.

Bir sistemle ilgili birden çok model oluşturulabilir. Tek bir model ile sistemin tamamını görmeye çalışmak yerine, üzerinde çalışılan sistemin farklı yönlerini öne çıkaran modeller hazırlanabilir. Örneğin inşaat firması toplu konutu müşterilere anlatmak üzere estetik tasarımı ön plana çıkan bir maket hazırlarken, bu toplu konut projesindeki binaların elektrik tesisatı için farklı, su tesisatı için farklı, genel daire görünümü için farklı projeler hazırlar. Böylece aynı sistemin farklı yönleriyle ilgilenen kimseler, yalnızca kendilerini ilgilendiren yönü öne çıkaran model üzerinde çalışma olanağı bulurlar.

1.2 Bilgisayar Programları ve Modelleme

Bilgisayar programları, makina ortamına aktarılacak problemin çözümünü oluşturmak üzere geliştirilir. Bilgisayar programı yazmak için öncelikle, doğal dil ile ifade edilen problemin makina ortamında yeniden oluşturulması gerekir. Çoğu zaman geliştirilecek program gerçek sistemin tamamı ile ilgilenmez. Bu durumda, üzerinde çalışılan problem, sistemin belirli bir açıdan görünümü olarak değerlendirilebilir. Bilgisayar ortamına

bu görünümü aktarabilmek için bir model oluşturulması gerekir. Oluşturulan model gerçek hayattakine ne kadar benzer ya da yakınsa, programlamanın o kadar kolaylaşacağı düşünülür. Çünkü programcı da bir insandır ve her ne kadar programlama yaklaşımlarına hakim ve bu doğrultuda düşünmeyi öğrenmiş olsa da içinde yaşadığı dünyayı olduğu gibi algılamak, üzerinde çalıştığı problemi başka insanların gördüğü gibi görmek onun için de en doğal olanıdır.

1.3 Nesneye Yönelik Modelleme

Nesneye yönelik programlama yaklaşımı, gerçek hayattan alınmış problemi çözmek üzere oluşturulacak modelin, gene gerçek hayatta var olan nesneler ve bu nesneler arasındaki ilişkilerden faydalanılarak oluşturulmasını ilke edinmiştir. Problem aynen gerçek hayatta görüldüğü şekli ile sanal ortama aktarılabilirse, günlük yaşamında nesneler ve nesneler arası ilişkiler ile etkileşimde olan programcı, nesneye yönelik model sayesinde, üzerinde çalıştığı problemi aynen gerçek hayatta olduğu şekliyle görebilecektir.

Peki nesneye yönelik model nasıl oluşturulur? Problem gerçek hayattakine benzer şekilde nasıl modellenebilir? Bu sorulara birlikte yanıt arayalım.

1.3.1 Sınıf ve Nesne Kavramları

Bir otomobil düşünelim. Otomobilin marka ve modeli ne olursa olsun, gaza basınca hızlandığını, frene basınca yavaşladığını, direksiyonu herhangi bir tarafa çevirince otomobilin o tarafa döndüğünü hepimiz biliyoruz. Bunlar otomobillerin genel **davranışlarıdır** ve bütün otomobiller bu davranışları sergiler.

Ayrıca, her otomobilin bir motoru, lastikleri, farları, direksiyonu, dikiz aynası vardır. Marka ve modeli ne olursa olsun, gene bütün otomobillerde bunlar ve daha birçok başka aksam bulunmaktadır. Başka bir deyişle, bütün otomobiller bu **özellikleri** taşımaktadır.

Bu örnekte otomobil bir sınıftır. “Otomobil” denilince aklımıza gelen temel davranışlar ve özellikler, bütün otomobillerde vardır. Her otomobil gaza basıldığında aynı sürede 100 km hıza ulaşamıyor olsa da, *gaza basıldığında bütün otomobiller hızlanır*. Ya da bütün otomobillerin lastikleri aynı büyüklükte olmasa da, *bütün otomobillerin lastikleri vardır*. Sınıf, aynı özellik ve davranışları sergileyen varlıkların ortak kümesini belirleyen tanımdır. Hiç otomobil görmemiş birisine otomobilin ne olduğunu anlatmaya kalksak, kapımızın önünde duran belirli bir otomobili değil, o kişinin bir otomobil gördüğünde tanımasını sağlayacak, bütün otomobiller için ortak olan bilgiyi aktarmaya çalışırız.

Başka bir örnek verelim; “kalem”. Kalem dediğimizde hepimizin aklına birşeyler gelir. Yazı yazmak üzere kullanıyoruz, çeşitli renk, boy, tip, uç kalınlığı ve fiyatlarda olabilir.. vs. “Kalem” deyince, hepimizin tanıdığı, bildiği birşeyden bahsetiğimiz için hepimiz kafamızda birşeyler canlandırıyoruz. Ancak çok büyük olasılıkla herkesin aklında canlanan resim birbirinden farklı; kimimiz kurşun kalem, kimimiz dolmakalem, kimimiz tahta kalemi düşünüyoruz ve belki kurşun kalemlerin çoğunun tipi birbirinden farklı. Ama sonuçta bütün kalemlerin yazı yazma **davranışı** ve renk ya da uç kalınlığı **özelliği** var ve bunlar “kalem” sınıfını belirleyen noktalar.

Özetle sınıf, o sınıftan olan bütün varlıkların ortak özellik ve davranışlarını anlatan bir tanımdır.

Nesne ise ait olduğu sınıftan gelen özelliklerin değerlerinin belli olduğu, sınıfı için tanımlı olan davranışları nasıl sergilediğinin bilindiği, somut olarak var olan, biricik bir kimliği olan varlıktır. Örneğin, otomobil sınıfına ait olan 06-XYZ-1234 plakalı bir otomobil nesnesinden bahsediyorsak, söz konusu nesnenin kimliği plaka numarasıdır ve genel olarak otomobillerden değil, markası, modeli, rengi belli olan, gaza basıldığında 100 km/saat hıza kaç saniyede ulaştığı bilinen, elle gösterilebilen tek bir varlıktan söz ediyoruz demektir.

Aynı sınıfa ait birçok nesne olabilir, şu an Ankara'da binlerce otomobil bulunduğu gibi. Aynı sınıfa ait olan nesnelerin hepsinde, o sınıftan gelen özellik ve davranışlar bulunmaktadır. Ancak herbir nesne için bu özelliklerin değerleri farklı, davranışların gerçekleştirilişi de bu özelliklere bağlı olarak farklı olabilir. Örneğin, 06-ZBC-9876 plakalı kırmızı spor otomobil 100 km/saat hıza 6 saniyede ulaşırken, 06-XYZ-1234 plakalı mavi otomobil 100 km/saat hıza 12 saniyede ulaşıyor olabilir. Ya da spor otomobilimiz 100 km'lik yolda 10 lt benzin tüketirken, diğer otomobilimiz aynı mesafede 6,5 lt benzin tüketiyor olabilir. Burada önemli olan, her iki otomobilin de otomobil sınıfından gelen özellik ve davranışları kendilerine özgü bir biçimde sergiliyor olmalarıdır.

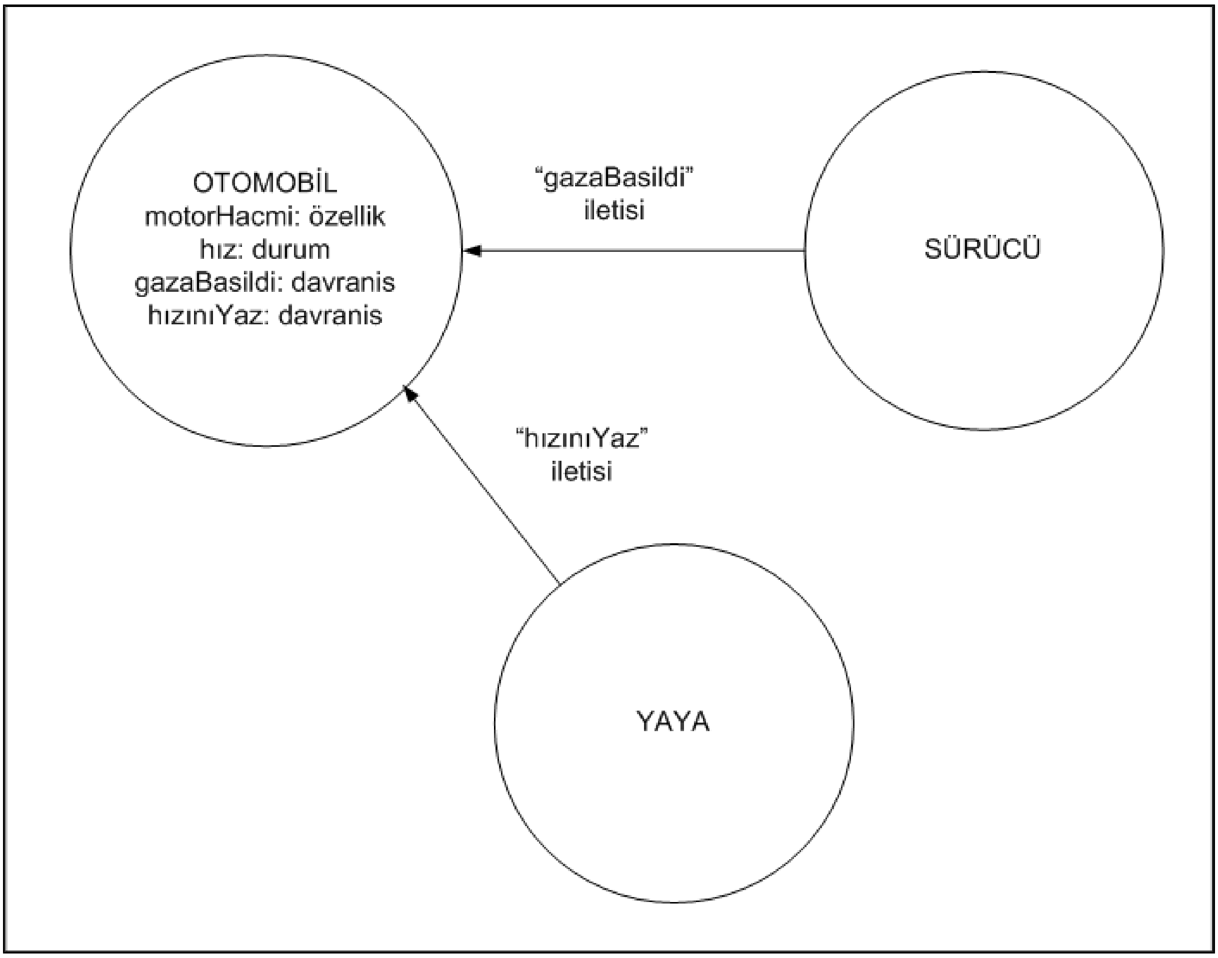
1.4 Nesneye Yönelik Programlama

Nesneye yönelik programlama, nesnelerin birbirlerine ileti göndermeleri ilkesine dayanır. Bir nesne, başka bir nesneye bir **ileti (*message*)** göndererek, o nesneden bir **davranış (*behaviour*)** sergilemesini ister. İletiyi alan nesne, bu iletiye göre davranışını gerçekleştirir ve kendi **durum (*state*)** bilgisini değiştirir.

Bunu bir örnekle açıklamaya çalışalım:

Bir otomobil, otomobilin sürücüsü ve otomobilin dışındaki bir yaya düşünelim. Burada üç tane nesnemiz var; OTOMOBİL, SÜRÜCÜ ve YAYA nesneleri. Otomobilin "gazaBasıldı" **davranışı** ile "hız" ve "motorHacmi" **özellikleri** var. Ayrıca, otomobilin "hızınıYaz" davranışı da, bu davranış sergilendiği anda "hız" özelliğinin değerini ekrana yazıyor. Sürücü ve yaya nesnelerinin özellik ve davranışları ile ise şu anda ilgilenmiyoruz.

Şekil 2.1'de, bu nesneler görülüyor.



Şekil 1-1. Nesneye Yönelik Programlama

İlk olarak SÜRÜCÜ nesnesi, OTOMOBİL nesnesine bir ileti göndererek "gazaBasildi" davranışında bulunmasını istiyor. Bunun üzerine, OTOMOBİL nesnesi, "hız" özelliğinin değerini "motorHacmi" özelliğinin değerine bağlı olarak değiştiriyor (hızlanıyor). Yani OTOMOBİL, **davranışı gerçekleştirerek durumunu değiştiriyor**. Bu işlem bittikten sonra, YAYA nesnesi, OTOMOBİL nesnesine yeni bir ileti göndererek "hızınıYaz" davranışında bulunmasını istiyor. "hızınıYaz" davranışı, OTOMOBİL nesnesinin yeni hızını ("hız" özelliğinin değerini) ekrana yazıyor.

Eğer bu nesneleri, özellikleri ve davranışları daha ayrıntılı bir şekilde ele alırsak; örneğin OTOMOBİL nesnesinin "hızınıYaz" davranışını ekrana yazı olarak değil de sürekli yenilenen bir resim olarak getirmeyi başarabilirsek; bir otomobil yarışı oyunu programlayabiliriz.

1.5 Nesneye Yönelik Çözümleme ve Tasarım

Çözümleme, hedef sistemin **ne yapması gerektiğinin** belirlendiği aşamadır. Bu aşamada çalışan insanın bilgisayar programlama bilgisinin olması gerekmez. Önemli olan, üzerinde çalıştığı sistemi tanıyabilmek, dile getirilen gereksinimleri iyi anlayıp bu gereksinimlerin arkasında kalan ve dile getiril(e)meyen gereksinimleri de bulup çıkartmak, işin kurallarını ve işleyişini irdeyebilmektir.

Nesneye yönelik çözümleme, çözümlemeyi yaparken sınıflar ve nesnelerden faydalanır. Sistemdeki sınıfların, bu sınıfların özellik ve davranışlarının ne olması gerektiğinin belirlenmesi ile uğraşır.

Tasarım ise hedef sistemin **nasıl gerçekleştirileceği** sorusuna yanıt arar. Tasarım aşaması modelin ortaya çıktığı aşamadır. Model tasarıma bağlı kalınarak oluşturulur ve çözümleme aşamasında belirlenen gereksinimlerin karşılanıp karşılanmadığı model üzerinde sınanabilir.

Nesneye yönelik tasarım, modeli oluşturmak üzere sınıflar ve nesnelerden faydalanır. Sınıflar arası ilişkiler ayrıntılı olarak incelenir, gereksinimleri karşılamak üzere ilişkiler, özellikler ve davranışlar şekillendirilir. Tasarım sona erdiğinde, ortaya nesneye yönelik bir model çıkar. Nesneye yönelik modelin ifade edilmesi için bir takım görsel diller kullanılmaktadır. Bunlardan en bilineni ve artık bir standart haline gelmiş olan dil, Unified Modelling Language (UML) adı ile bilinir. Tasarım aşaması genelde hedef sistemin çeşitli açılardan görünüşlerini ifade eden UML belgeleri ile son bulur.

1.6 Nesneye Yönelik Programlama Dilleri

Nesneye yönelik model ortaya çıktıktan sonra, bu modelden nesneye yönelik programa geçmek modeli oluşturmaktan daha kolaydır. Çünkü programlama aşaması, modelin bir programlama dili ile ifade edilmesi aşamasıdır ve modeldeki herşeyin programlama dilinde neye karşılık geldiği bellidir. Seçilen herhangi bir nesneye yönelik programlama dili ile o model (modeldeki bazı alt düzey ayrıntılar dışında) kodlanabilir.

Eğer UML gibi bir dil kullanılmışsa, oluşturulan model bir UML aracı ile çizilebilir. Çoğu zaman bu tür araçlar, seçilen programlama diline göre kod üretebilme (*code generation*), hatta verilen koddan modele geri dönebilme (*reverse-engineering*) yeteneklerine sahiptir.

Demek ki önemli olan hedef sistemin hangi programlama dili ile geliştirileceği değil, hedef sistem için gerekli olan modelin oluşturulabilmesidir. Sonraki aşamada programalama dili seçimi programlama ekibinin bilgi birikimine, kurumun daha önceki yatırım ve deneyimlerine ya da sistemin geliştirilmesini isteyen müşterinin özel gereksinimlerine göre yapılabilir.

1.7 Özet

Bu bölümde, Nesneye Yönelik Programlama ile ilgili çeşitli kavramlara kısaca değindik.

Model, hedef sistemin tamamını bir kerede anlamaya çalışmak yerine ilgilenilen kısmını daha kolay anlayabilmek/anlatabilmek için çok uzun zamandır kullanılmaktadır. Modelleme ise modeli oluşturma işlemidir.

Bilgisayar programları, gerçek hayattaki problemlerin bilgisayarın doğru ve hızlı işlem yapma yeneteğinden faydalanılarak çözülmesi amacıyla geliştirilirler ve bunun için öncelikle bilgisayar ortamında problemin ifade edilebilmesi gerekir. Bu süreçte modeller kullanılır.

Nesneye yönelik modelleme, modelleme işleminin nesneler ve nesneler arası ilişkilerin incelenmesi yolu ile yapılması ilkesine dayanır.

Nesneye yönelik programlama, nesnelerin birbirlerine ileti göndermeleri, iletiyi alan nesnenin bir davranışı yerine getirmesi ve davranışın yerine getirilmesi sonucunda durumunun değişmesi ile gerçekleştirilir.

Nesneye yönelik çözümleme, sistemde ne yapılması gerektiğinin ve hangi sınıf, nesne, özellik ve davranışların bulunduğunun belirlendiği aşamadır.

Nesneye yönelik tasarım ise sınıflar arası ilişkilerin belirlendiği, özellik ve davranışların yeniden incelenip son şeklini aldığı, sonucunda nesneye yönelik modelin ortaya çıktığı aşamadır.

Nesneye yönelik programlar, nesneye yönelik modelin herhangi bir nesneye yönelik programlama dili ile ifade edilmesi sonucunda ortaya çıkar. Bir model birçok farklı dille ifade edilebilir.

2 Java Platformu

Java yalnızca bir programlama dili değildir. Java Platformunu oluşturan çeşitli alt ögeler bulunmaktadır:

- Bir programlama dili,
- Bu programlama dili ile yazılmış programların üzerinde çalışacağı altyapı,
- Farklı alanlarda (masaüstü, cep telefonu, saat, web tabanlı...) uygulama geliştirmek için kullanılan yardımcı araç ve kütüphaneler,
- Bu kütüphanelerin belirtimleri (*specification*),
- Bütün bunların en iyi başarımı elde etmek üzere nasıl kullanılmaları gerektiğini anlatan en iyi uygulama (*best practice*) belgelerinin tamamı

Java Platformu, birbirlerinden tamamen bağımsız olmayan üç alt platform/teknoloji içermektedir;

Java Standart Edition (JSE): Temel Java uygulamalarının geliştirilmesi için gerekli çekirdektir.

Java Enterprise Edition (JEE): Büyük ölçekli, dağıtılmış kurumsal uygulamaların geliştirilebilmesi için kullanılır.

Java Micro Edition (JME): Cep telefonu, saat, el bilgisayarı gibi küçük ölçekli ortamlara uygulama geliştirmek için kullanılır.

Herbir teknoloji kendi içinde birçok kütüphane, uygulama çatısı (*framework*) ve belge içermektedir. Ancak öncelikle *JSE* platform/teknolojisinin öğrenilmesi zorunludur. Çünkü Java platformlarından herhangi birisi ile uygulama geliştirebilmek için öncelikle *Core (Çekirdek) Java* adı verilen kısım bilinmelidir.

Bu kitabın ilgi alanı *JEE* ya da *JME* platformları değildir. Kitap içerisinde "Java Platformu" denildiğinde, aksi belirtilmedikçe, *JSE*'den bahsedilmektedir.

"Platform", bir programın çalıştığı donanım ya da yazılım ortamıdır.

"Java platformu", Java programlarının çalışması için gerekli olan iki bileşenden oluşur:

1- Java Sanal Makinesi (Java Virtual Machine)

2- Java Uygulama Programlama Arayüzü (Application Programming Interface-API)

"Platformdan Bağımsızlık" bağlamında kullanılacak olan "platform" sözcüğü ise, donanım ve o donanım üzerinde çalışacak olan işletim sistemini ifade etmektedir. Örneğin Windows32 platformu, 32 ikillik Windows işletim sisteminin çalıştığı bir ortamı ifade etmektedir.

Platform sözcüğünün kullanıldığı bağlama göre değerlendirilmesi gereklidir.

2.1 Platformdan Bağımsızlık

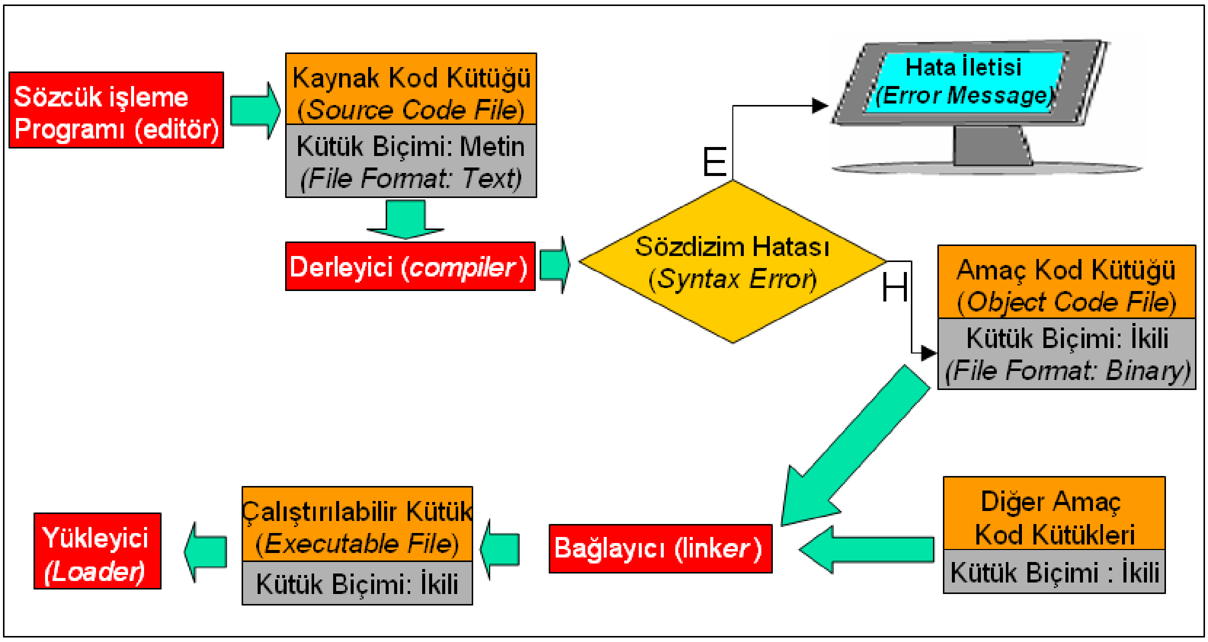
Java ile ilgili konuşulurken genelde ilk söylenen Java'nın platformdan bağımsız bir dil olduğudur. Peki "platformdan bağımsızlık" ne demektir?

Platformdan bağımsızlığın ne olduğunu anlatabilmek için, öncelikle platforma bağımlılığın ne olduğunu açıklamaya çalışalım. Üzerinde Windows98 işletim sisteminin çalışmakta olduğu bir kişisel bilgisayar üzerinde, Visual Studio 6.0 tümleşik geliştirme ortamını (*Integrated Development Environment - IDE*) kullanarak C dili ile bir program geliştirdiğimizi düşünelim. Program derlendiği zaman, çalıştırılabilir (üzerine çift tıkladığımızda ya da konsolda adını yazdığımızda çalışmaya başlayan: *executable*), .exe uzantılı bir dosya (*file*) oluşur. Bu program, Windows98 işletim sistemi üzerinde çalıştırılabilir bir dosya (*executable file*) olarak saklanmaktadır.

Bu çalıştırılabilir dosyayı, üzerinde Linux işletim sistemi çalışan başka bir makineye kopyaladığımızda programın çalışmadığını görürüz. Çünkü o çalıştırılabilir dosya, ancak Windows ortamında çalışabilecek şekilde oluşturulmuştur. Başka bir deyişle, *hangi platformda çalışacağı bellidir*. Bunun nedeni, belli bir platform (işletim sistemi + donanım) üzerinde

çalışmak üzere oluşturulmuş çalıştırılabilir dosya içinde saklanan bilgilerin o platforma özel olarak üretilmiş olmasıdır.

Bir C programının geçirdiği aşamalar Şekil 4-1'de görülmektedir.



Şekil 2-1. C Programlarının Geçirdiği Aşamalar

C programları platforma özel olduklarından, aynı C programının başka bir platformda çalıştırılabilmesi, C programının o platform için yeniden derlenmesini ve çalıştırılabilir dosyaya dönüştürülmesini gerektirir.

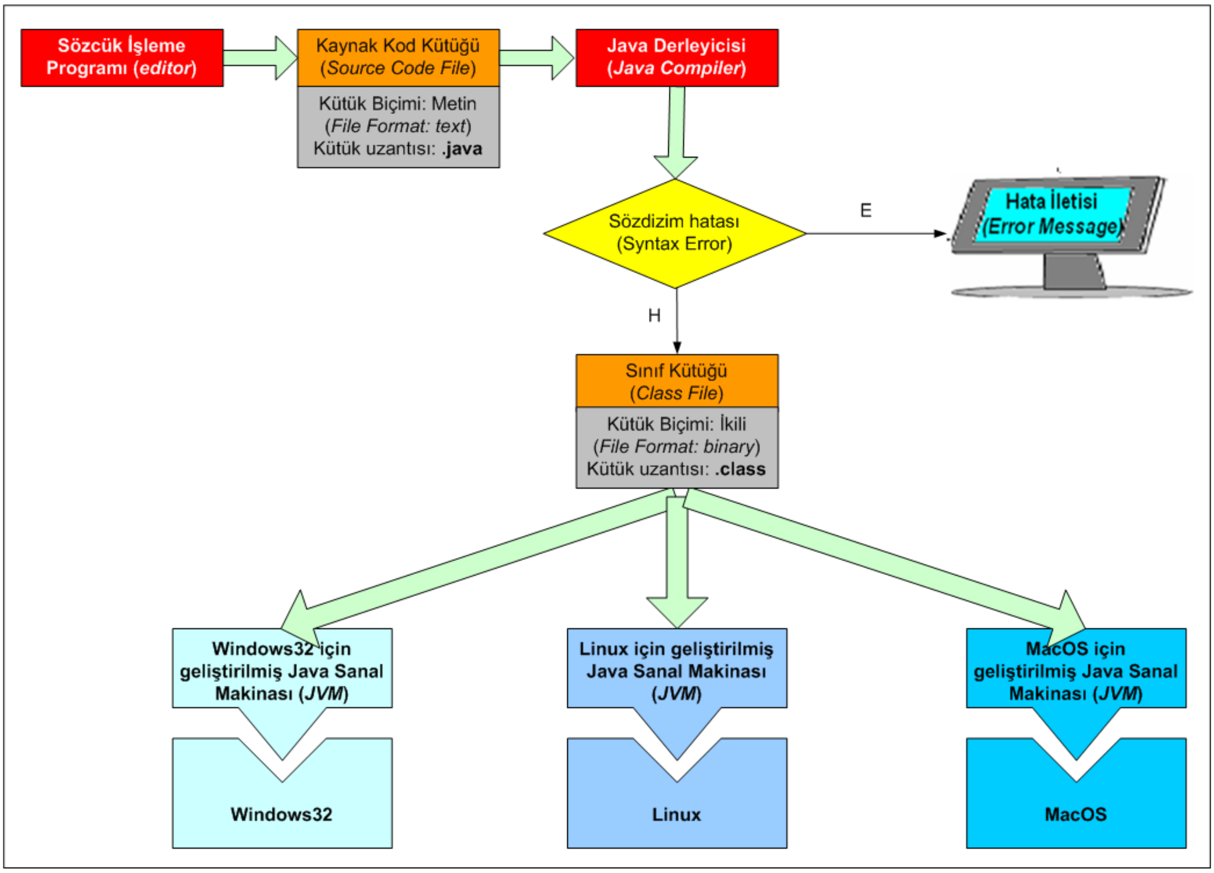
Java platformunda ise yaklaşım farklıdır. Programın üzerinde çalışacağı platforma özel kodlar üretilerek çalıştırılabilir dosya oluşturulması yerine, herhangi bir platform üzerinde başka bir "ara uygulama" tarafından yorumlanabilecek kodlar üretilir. Bu "ara uygulama" ise Java Sanal Makinesi (*Java Virtual Machine – JVM*) adı ile bilinir.

Java kaynak kodları yazılıp **`.java`** uzantısı ile kaydedildikten sonra, bir Java derleyicisi ile derlenir. Derleme sonucunda oluşan **`.class`** uzantılı dosyaya **sınıf dosyası** (*class file*) adı verilir. Bir sınıf dosyası, herhangi bir platforma özel olmayan, herhangi bir JVM tarafından yorumlanabilecek, *byte-code* olarak adlandırılan ikil (*binary*) bilgiler içerir. Örneğin WindowsXP ortamında yazdığımız Java programını Linux ortamına taşımak istediğimizde, bu derlenmiş sınıf dosyalarını (.class uzantılı dosyalarını)

Linux ortamına kopyalamamız yeterlidir. Linux ortamı için geliştirilmiş olan JVM, üzerinde hiçbir değişiklik yapmadan Linux ortamına aktardığımız dosyaları yorumlayarak çalıştıracaktır.

Başka bir deyişle, Java dilinin platformdan bağımsız olması, bütün yaygın platformlar için (Windows, Linux, Unix ya da Apple ortamları için) birer JVM'nin geliştirilmiş olması ile sağlanmıştır. JVM, bir belirtime (*specification*) uymaktadır ve bu sayede o belirtime uygun olarak geliştirilmiş bütün Java programlarını farklı platformlarda aynı şekilde çalıştırabilmektedir. Kısaca, Java kaynak kodu ile platforma özel makine kodu arasına bir katman daha eklenmiş, derlenmiş Java dosyalarının bu katmandan alınan hizmet ile çalıştırılması sağlanmıştır. Böylece programcıların işi kolaylaşmakta, geliştirdikleri uygulamalar hiç değişikliğe uğramadan farklı platformlarda çalışabilmektedir.

Şekil 4-2'de bir Java programının platformdan bağımsızlığının sağlanmasında JVM'nin nasıl rol oynadığı görülmektedir.



Şekil 2-2. Java Programlarının Platformdan Bağımsızlığında JVM'nin Rolü

Java'nın platformdan bağımsızlığının bu şekilde sağlanması, yani java kaynak dosyalarından üretilen kodların (*byte-code)* JVM gibi bir ara katman tarafından işletilmesi, Java programlarının, hedef platforma özel üretilen özgün kodların (*native codes*) işletilmesi mantığına dayanan C gibi dillerle geliştirilen programlara oranla az da olsa daha yavaş olmasına neden olmaktadır. Bu, platformdan bağımsızlık için ödenen bir bedel olarak görülebilir. Bununla birlikte derleyici ve JVM teknolojilerindeki ilerlemeler Java programlarının başarımını platformdan bağımsızlık prensibinden ödün vermeden özgün kodların başarımına yaklaştırmaktadır.

Java programlarının platformdan bağımsızlığını vurgulayan slogan:

"Write once, run anywhere: Bir kez yaz, heryerde çalıştır"

2.2 Java Sanal Makinesi (*Java Virtual Machine - JVM*)

Java Sanal Makinesinin, Java programlarının paltformdan bağımsızlığını nasıl sağladığını bir önceki bölümde açıklamaya çalıştık. Bu bölümde ise JVM'nin Java programlarını nasıl işlettiğini kabaca anlatmaya çalışacağız.

Herşeyden önce şunu söylemekte yarar var: birden fazla JVM olabilir ve bunların birbirlerine göre daha iyi ya da daha kötü başarımları (*performance*) olabilir. Bunun nedeni basittir. JVM için bir belirtim (*specification*) bulunmaktadır. Bu belirtim, JVM adı verilen yazılımların uymaları gereken kuralları, sağlamaları gereken kısıtları, kısaca bir yazılımın JVM olarak adlandırılabilmesi için gerekli herşeyi tanımlamaktadır. Belirtim, "ne" olması gerektiğini belirleyen ancak "nasıl" olması gerektiği konusunda bir kısıt getirmeyen bir belgedir. Dolayısıyla, aynı belirtimi gerçekleştiren farklı yazılım ekipleri, kendi gerçekleştirimlerinde (*implementation*) kullandıkları veri yapıları ya da algoritmalar sayesinde diğerlerinden daha yüksek başarımlı JVM'ler geliştirebilirler.

http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html adresinde JVM belirtimi bulunabilir.

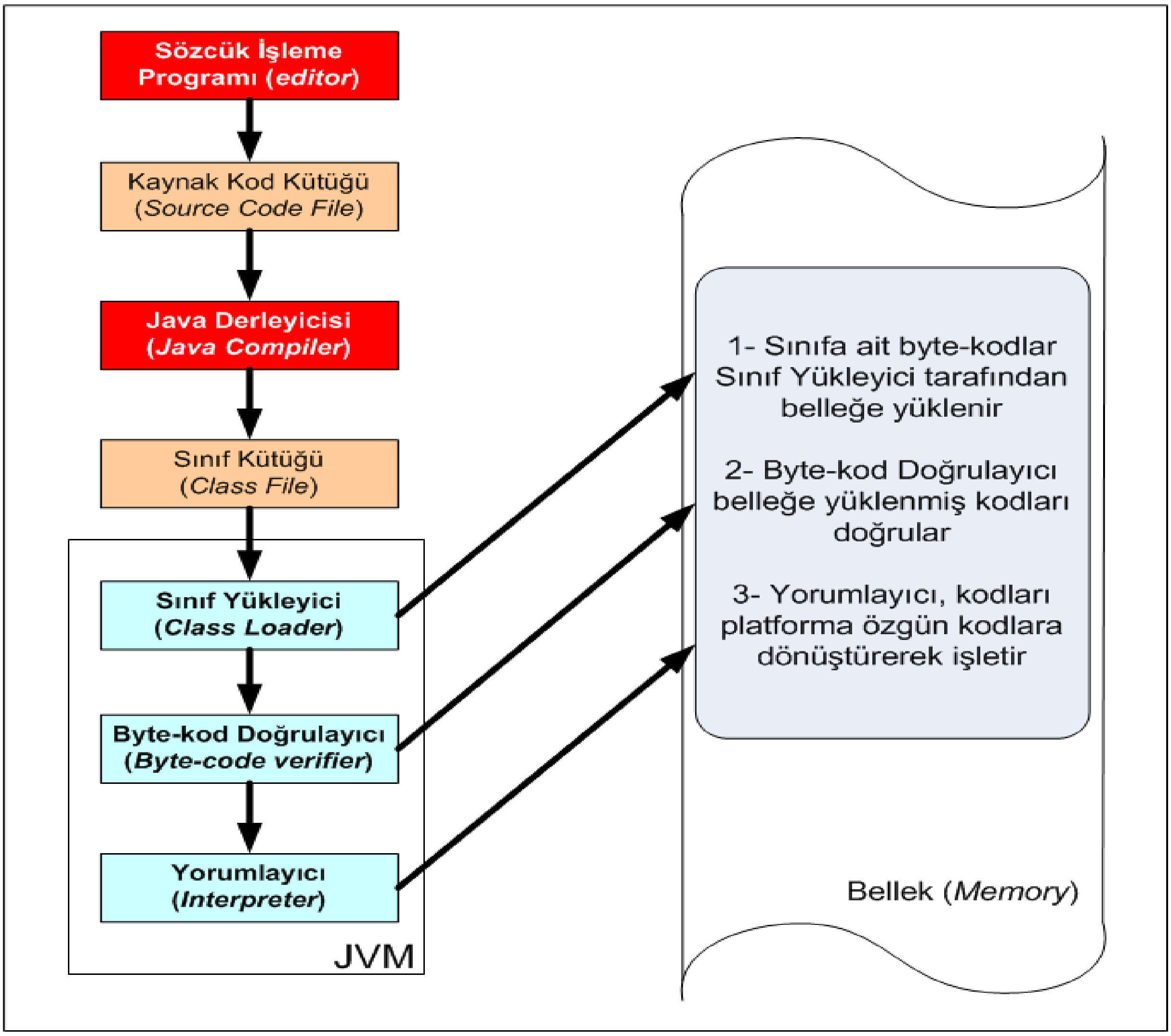
Java Sanal Makinesi, Java programlarını çalıştırmak üzere bir takım görevler içermektedir. Bu görevlerin bazılarına kısaca bakalım:

Sınıf Yükleyici (*Class Loader*): Java programları birbirleriyle etkileşimde bulunan sınıflardan oluşur. Bir sınıfın nesnesinin oluşturulabilmesi için önce sınıfın belleğe yüklenmesi gerekir. Sınıf Yükleyici görevi, `.class` uzantılı dosya içinde bulunan Java sınıfını belleğe yükler.

Byte-kod Doğrulayıcı (*Byte-code Verifier*): Belleğe yüklenen byte-kodları inceleyerek türlere dair bilgileri oluşturur ve bütün parametrelerin türlerini doğrular. Ayrıca bir takım güvenlik kısıtlarını denetler.

Yorumlayıcı (*Interpreter*): Belleğe yüklenmiş ve doğrulanmış byte-kodları JVM'nin üzerinde çalışmakta olduğu platforma özel kodlara dönüştürerek işletir.

Şekil 4-3'te bu görevlerin çalışması görülmektedir.



Şekil 2-3. JVM İçindeki Bazı Görevler

Java programlarının çalıştırılmasında rol alan 3 temel göreve baktıktan sonra, biraz da JVM'nin yapısına bakalım. Daha sonra anlatacağımız kesimlerde bu yapıya göndermeler yapacağız.

Java Sanal Makinesinin, "sanal donanım" kesimi 4 temel bölümden oluşur: yazmaçlar (*registers*), yığıt (*stack*), yığın (*heap*) ve yöntem alanı (*method-area*). Bu bölümler, oluşturdukları sanal makine gibi sanal (ya da soyut) bölümlerdir ancak her JVM gerçekleştiriminde bir şekilde bulunmak zorundadırlar.

32 ikillik JVM içinde bir bellek adresi 32 ikil (*bit – binary digit*) ile ifade edilir. Başka bir deyişle 32 ikillik JVM, herbiri 1 byte veri içeren 2^{32} adet bellek yerleşimini, yani 4GB'lik belleği adresleyebilir.

Adresleme hesaplarının nasıl yapıldığını merak eden okuyucularımız için hesaplamayı biraz daha açalım:

Adreslemede kullanılan ikil sayısı	Bu ikiller ile ifade edilebilecek farklı değer sayısı	Bu sayının ifadesinde kullanılan kısaltma	Bu kadar Byte anlamında kullanılan kısaltma
10 ikil	$2^{10} = 1024$	Kilo	KB
20 ikil	$2^{20} = 2^{10} \times 2^{10}$ = Kilo x 1024	Mega	MB
30 ikil	$2^{30} = 2^{20} \times 2^{10}$ = Mega x 1024	Giga	GB
32 ikil	$2^{32} = 4 \times 2^{30}$ = 4 x Giga	4 Giga	4GB

Tablo 2-1. Adresleme Hesapları

32 ikillik JVM'nin içindeki her yazmaç 32-ikil uzunluğunda bir adres tutar. Yığıt, yığın ve yöntem alanları da bu 32-ikil ile ifade edilebilen 4GB büyüklüğündeki belleğin bir yerlerinde olmak zorundadır. Ancak neyin nerede saklanacağı, kullanılan JVM'ye özeldir.

Burada anlatılanlar kabaca şu anlama gelmektedir. Kişisel bilgisayarınızda bir Java programı işletmek için bilgisayarınıza kuracağınız JVM (sonraki kesimlerde JVM'nin nasıl kurulduğuna bakacağız), bilgisayarınızın belleğinin en fazla 4 GB'lık bir kısmını kullanabilir. Günümüzde satılmakta olan kişisel bilgisayarların genelde 2GB - 4GB bellekleri olduğunu düşünürsek, bu sınırın çoğu zaman yeterli olacağı sonucunu çıkarabiliriz.

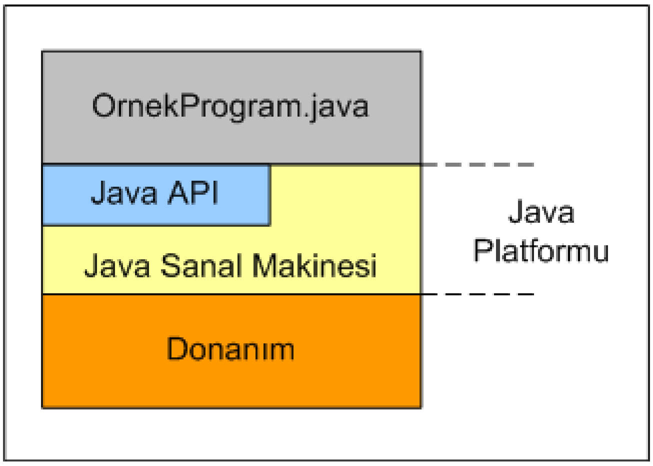
2.3 Java Uygulama Programlama Arayüzü (*Java Application Programming Interface - API*)

Uygulama Programlama Arayüzü, çeşitli yetenekler sunan hazır yazılım bileşenlerinden oluşan bir derlemdir.

Java API, çeşitli amaçlarla kullanılmak üzere tasarlanmış çok zengin bir bileşen kütüphanesi sunar. Java ile uygulama geliştirilirken bu bileşenlerden faydalanılır. Örneğin, eğer dosya işlemleri yapan bir program yazmak istiyorsak, Java API'de bulunan ve dosyalarla ilgili her türlü işlemi

yapmakta kullanılan sınıf ve arayüzleri bilmemiz gerekir. Ya da kolay kullanımlı ve zengin bir grafik kullanıcı arayüzü (*Graphical User Interface - GUI*) olan bir uygulama için *Java Swing API*'yi kullanabiliriz.

Şekil 3-4'te JVM ve Java API birlikte görülmektedir.



Şekil 2-4. Java Sanal Makinası, Java API ve Java Programı

Java platformunu tanıdıktan ve Java ile nesneye yönelik uygulama geliştirme yaklaşımını öğrendikten sonra, kendisini geliştirmek isteyen bir kimsenin yapması gereken iş Java *API*'yi öğrenmek olacaktır. Bileşenlerin tamamının öğrenilmesi çok olası olmasa da, neler bulunduğuna dair fikir sahibi olmak önemlidir. Böylece herhangi bir programlama gereksinimini karşılamak üzere nelerin kulllanılabileceği bilinmiş olur ve çözüme ulaşmak kolaylaşır.

Java API'de neler olduğuna dair fikir edinmek için en kolay yöntem herhalde ***Java API Documentation*** adı ile bilinen ve bu kitabın yazılmakta olduğu zamanda http://java.sun.com/javase/6/docs/api/ adresinden erişilebilen HTML belgelerine bir gözatmaktır.

> ***Java API Documentation, www.java.sun.com adresinden indirilebilir. Java ile uygulama geliştirirken en sık başvurulacak belge olduğundan, bu belgeyi bilgisayarınıza indirmenizde fayda vardır.***

Şekil 3-5'te *Java API Documentation* belgelerinden örnek bir görünüm bulunmaktadır. Görüldüğü gibi sayfa 3 çerçeveden oluşmaktadır. Sol üst taraftaki çerçevede *API*'de bulunan paketler (*packages*), sol alt çerçevede

seçilen pakette bulunan sınıf ve arayüzler, sağ taraftaki büyük çerçevede ise sol alt çerçeveden seçilen sınıf ya da arayüz ile ilgili açıklamalar ve o sınıf ya da arayüzün yöntemleri ile bunlara ait ayrıntılı bilgiler bulunmaktadır.



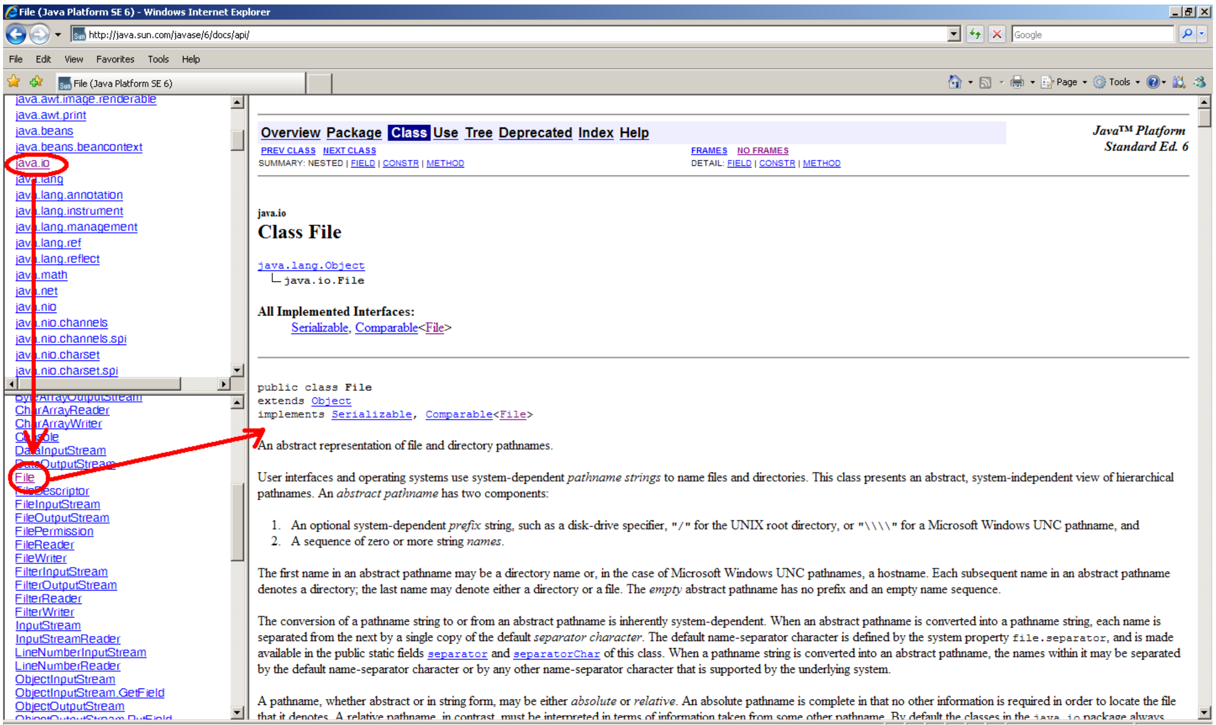
Şekil 2-5. Java API Documentation Belgelerinden Örnek Bir Görünüm

Belgenin kullanılışını bir örnekle açıklamaya çalışalım. Diyelim ki dosya işlemlerine gereksinim duyduğumuz bir uygulama geliştiriyoruz. Diskteki bir dosyayı açıp içindeki veriler üzerinde işlem yapmamız gerekiyor. Bununla ilgili olarak ne kullanabileceğimiz hakkında az da olsa fikrimiz var ancak neyi nasıl kullanacağımızı tam olarak bilmiyoruz.

Bu durumda, *Java API Documentation* belgelerine başvuruyoruz. Sol üst çerçeveden `java.io` paketini seçiyoruz. Bu paket, Java'da giriş/çıkış (*input/output*) işlemlerinin yapılabilmesi için kullanılabilecek sınıf ve arayüzleri içermektedir. Sol alt çerçeveye, paketin içindeki sınıf ve arayüzlerin listesi gelir. `File` sınıfının üzerine tıkladığımızda, sağdaki çerçevede `File` sınıfına ait bilgiler gelir. Sınıfla ilgili genel açıklamaları okuduktan sonra, alt tarafta bu sınıfın içinde bulunan yöntemlerin listesini

inceleyebilir ve örneğin program içerisinden bir dosyanın adını değiştirmek için `renameTo` adlı bir yöntemin kullanılacağını görebiliriz.

Şekil 4-6'da bu adımlar görülmektedir.



Şekil 2-6. Dosya İşlemleri İçin `File` Sınıfının İncelenmesi

2.4 Özet

Java yalnızca bir programlama dili değil, bu dil ile yazılmış programların üzerinde çalıştırıldığı bir platformdur.

Java programlarının platformdan bağımsızlıkları, farklı platformlar için geliştirilmiş olan Java Sanal Makineleri ile sağlanmaktadır.

Java Sanal Makinesi, belli bir belirtime uygun olarak gerçekleştirilmiş bir yazılımdır. Sanal bir donanımı yönetir ve (32 ikillik) sanal donanım en fazla 4GB bellek kullanabilmektedir.

Java platformu, Java Sanal Makinesi ile Java Uygulama Programlama Arayüzü'nden oluşur. Uygulama Programlama Arayüzü, hazır yazılım bileşenlerinin bir derlemidir.

Java Uygulama Programlama Arayüzü, Java uygulamalarında kullanılabilecek sınıf ve arayüz tanımlarını içerir. Bunlarla ilgili bilgi, Java API Belgeleri’nden edinilebilir.

3 Java Dili İle İlgili Temel Bilgiler

Bu bölümde, Java dili ile ilgili temel bilgilere yer verilecektir. Okuyucunun temel programlama bilgisi olduğu kabul edilecek ve "değişken nedir?", "döngü nedir?"... gibi konular anlatılmayacak, yalnızca bu tip temel yapıların Java'da nasıl ifade edildiklerine hızlıca bakılacaktır.

Java dili sözdizimsel olarak C, C++, C# dillerine çok benzer. İşleçler, işleçlerin öncelikleri, değişken tanımlama ve ilk değer atama, denetim ve döngü yapıları büyük oranda aynıdır. Eğer bu dillerden herhangi birisini biliyorsanız, bu kesime kısaca bir gözatıp hızlıca geçebilirsiniz.

3.1 Sözdizim Kuralları (*Syntax Rules*)

Her programlama dilinde olduğu gibi Java'da da bir takım sözdizim kuralları vardır. Sözdizim kurallarına uygunluğu derleyici denetler ve kurallara uymayan durumları raporlar. Dolayısıyla sözdizim kurallarını tek tek açıklamaya çalışmak yerinde bir çaba olmayacaktır.

C, C++ ya da C# bilen okuyucular için Java'daki sözdizim kurallarının bu dillerdekilerle hemen hemen aynı olduğunu söyleyebiliriz.

3.1.1 Derleyiciler

Derleyiciler, kaynak kodu inceleyerek bu kodun programlama dilinin sözdizim kurallarına uygun olarak yazılıp yazılmadığını denetleyen, eğer dilin kurallarına uygun olmayan durumlar belirlenirse bunları raporlayarak programcıya kodu düzeltmesi için ipuçları veren programlardır.

3.1.2 ***Whitespace*** karakterleri

Kod yazılırken kullanılan bazı karakterler ***whitespace*** karakterleri olarak adlandırılır ve derleyici tarafından dikkate alınmaz:

end-of-line karakteri: **\n** ile ifade edilir; kod yazılırken **Enter** tuşuna her basılışta, imlecin bulunduğu yere bir ***end-of-line*** karakteri koyulur ve

kullanılan metin düzenleyici bu karakteri yorumlayarak imleci bir satır aşağıya indirip satır başına götürür.

`form-feed` karakteri: **`\f`** ile ifade edilir ve imleci sonraki sayfanın başına götürür.

`boşluk` karakteri: boşluk çubuğuna (*space-bar*) basıldığı zaman oluşan karakter, metin düzenleyici tarafından imlecin bir karakterlik boş yer bırakılarak ilerlemesini sağlar

`tab` karakteri: **`\t`** ile ifade edilir, çalışılan metin düzenleyicide genellikle 4 ya da 8 boşluk karakterine karşılık gelecek şekilde ayarlanabilir

`whitespace` karakterlerinin derleyici tarafından dikkate alınmaması şu anlama gelmektedir: programcı kodlama yaparken deyimler ya da değişkenler arasında istediği kadar boşluk bırakabilir, satır atlayabilir ya da **`tab`** tuşu ile ilerleyebilir, ancak sözdizim kurallarına aykırı bir iş yapmış olmaz. Bununla birlikte, ***`whitespace`*** karakterleri kodun daha okunur görünmesini sağlamak amacıyla kullanılır.

3.2 Açıklama Satırları (*Comment Lines*)

Açıklama satırları, kod içi belgeleme amacıyla kullanılan ve derleyici tarafından dikkate alınmayan kod kesimleridir. Bir programcının açıklama satırı yazmak için temel iki amacı olabilir:

Yazdığı kodun kritik kesimlerini açıklayarak, o koda daha sonra bakan kimselerin (büyük olasılıkla kendisinin) işini kolaylaştırmak: Aslında bu şekilde açıklama gerektiren kod, başarılı bir kod değildir. Çünkü zaten yazılan kodun hiçbir açıklamaya ihtiyaç duyulmadan kendisini açıklayabilmesi gerekir.

Yazdığı kod ile ilgili belge oluşturmak: Yazılımların belgelenmesi bir gerekliliktir. Bu belgeleme gereksinimi bazen sırf müşteri istiyor diye yerine getiriliyor olsa da, çoğu zaman ortaya çıkan ürünün yaşatılabilmesi ve hatta geliştirilebilmesi için gerçekleştirilmektedir. Belgeler, gereksinim

belirleme ve tasarım süreçlerinde raporlar şeklinde oluşturulurken, kodlama sürecinde kod içi belgeleme olarak ortaya çıkar.

Java'da açıklama satırları 3 farklı şekilde yazılır:

3.2.1 `//` ile yapılan açıklamalar

Tek satırlık bir açıklama yapılacaksa o satırın başına `//` işareti yazılır. Bir sonraki satıra geçilene kadar bu satıra yazılan herşey açıklama olarak değerlendirilir ve derleyici tarafından dikkate alınmaz. Daha doğru bir ifade ile; `//` işaretinden sonra satır sonuna kadar herşey açıklamadır. Anlaşılacağı üzere bu işaretin satırın en başında olması zorunlu değildir. Ancak kodlama alışkanlığı bakımından satır başında kullanılması daha uygundur.

Örnek:

```
// bu bir açıklama satırıdır
int sayi = 5;   // sayi değişkenine 5 değeri atandı.
```

Kod 3-1. Tek satıra yazılan açıklamalar

3.2.2 `/* ... */` ile yapılan açıklamalar

Eğer birden fazla satırda yazılan bir açıklama varsa, her satırın başına // işareti koymak programcıya zor gelebilir. Bunun yerine, açıklama olarak değerlendirilmesi istenen satırlar `/*` ve `*/` işaretleri arasına alınır. Bu iki işaret arasında kalan kesimler derleyici tarafından açıklama satırı olarak kabul edilir.

Örnek:

```
/* Bu birden fazla satırdan oluşan bir açıklamadır. Ancak bir açıklamanın
bu yolla ifade edilmesi için birden fazla satırdan oluşması zorunluluğu
yoktur. */
int sayi = 5;   /* sayi değişkenine 5 değeri atandı. */
```

Kod 3-2. Birden fazla satıra yazılan açıklamalar

3.2.3 /** ... */ ile yapılan açıklamalar

Bir uygulama geliştirilirken kod içi belgeleme yapmak güzel bir programlama alışkanlığıdır. Çünkü hem yapmakta olduğunuz işi en güzel o işi yaparken açıklayabilirsiniz, hem de açıklayabildiğiniz kodu anlamışsınız demektir ve o kodu açıklayarak yazdığınız için hata yapma olasılığınız düşer.

Öte yandan, çoğu zaman uygulamaların raporlarının oluşturulması gerekir. Kod yazıldıktan sonra kodun içine yazılan açıklamalardan bir belge oluşturarak bu belgeyi raporun sonuna eklemek programcının yükünü hafifletecektir. İşte şimdi bahsedeceğimiz üçüncü yöntem bu amaçla kullanılır. **/**** ve ***/** işaretleri arasına yazılan açıklamalar bir takım özel etiketler içerebilir. Kod içi belgeleme, bu etiketleri tanıyan ve etiketlerden faydalanarak belge üreten bir aracın yardımı ile belgeye dönüştürülebilmektedir.

Bu tarzda yazılan açıklama satırlarına Javadoc adı verilmektedir. Javadoc için kullanılabilecek bazı imler ve ne için kullanılabilecekleri aşağıda listelenmiştir:

Etiket	Açıklama
@author	Kodu yazan kişinin kim olduğunu belirtir
@deprecated	Yöntemin artık kullanımda olmadığını belirtir. Bazı uygulama geliştirme ortamları, bu şekilde işaretlenmiş yöntemler kullanıldığında programcıyı uyarır.
@exception	Bir yöntemin fırlattığı aykırı durumu belirtir
@param	Yöntemin aldığı parametreleri açıklamakta kullanılır
@return	Yöntemden dönen değeri açıklamakta kullanılır
@see	Başka bir yöntem ya da sınıf ile bağlantı kurar
@since	Bir yöntemin ne zamandan beri var olduğunu belirtir
@throws	Bir yöntemin fırlattığı aykırı durumu belirtir.
@version	Bir sınıf ya da yöntem için sürüm numarası belirtir.

Tablo 3-1. javadoc için kullanılan imlerden bazıları ve açıklamaları

javadoc.exe aracı, JDK'nın kurulduğu klasörde bin klasörünün altındadır. Bu araç, Java kaynak kodlarının üzerinden geçerek **/** .. */** işaretleri arasına yazılan açıklama satırlarından belge üretir. Belge, Java API Belgeleri ile ilgili kesimde anlatıldığı gibi HTML dili ile oluşturulmaktadır. Aslında Java API Belgesi'nin kendisi de bu yolla oluşturulmuş bir belgedir ve belki de Java API'sini kullanmak ve öğrenmek üzere kullanılabilecek en iyi ve kolay kullanımlı belgedir.

Örnek:

```
package ornek;
/**
 * @author Oğuz Aslantürk - 06.Mar.2007
 *
 */
public class AciklamaSatiriOrnegi {

      /**
       * Verilen sayının karekökünü bularak döndürür.
       * Sayının sıfırdan küçük olmadığını varsayar.
       *
       * @param sayi    Karekökü alınacak sayı
       * @return        Sayının karekökü
       */
      public double karekok(double sayi) {
            double kkok = 0;
            // burada karekök bulma algoritmasının çalıştığını kabul edelim
            return kkok;
      }
}
```

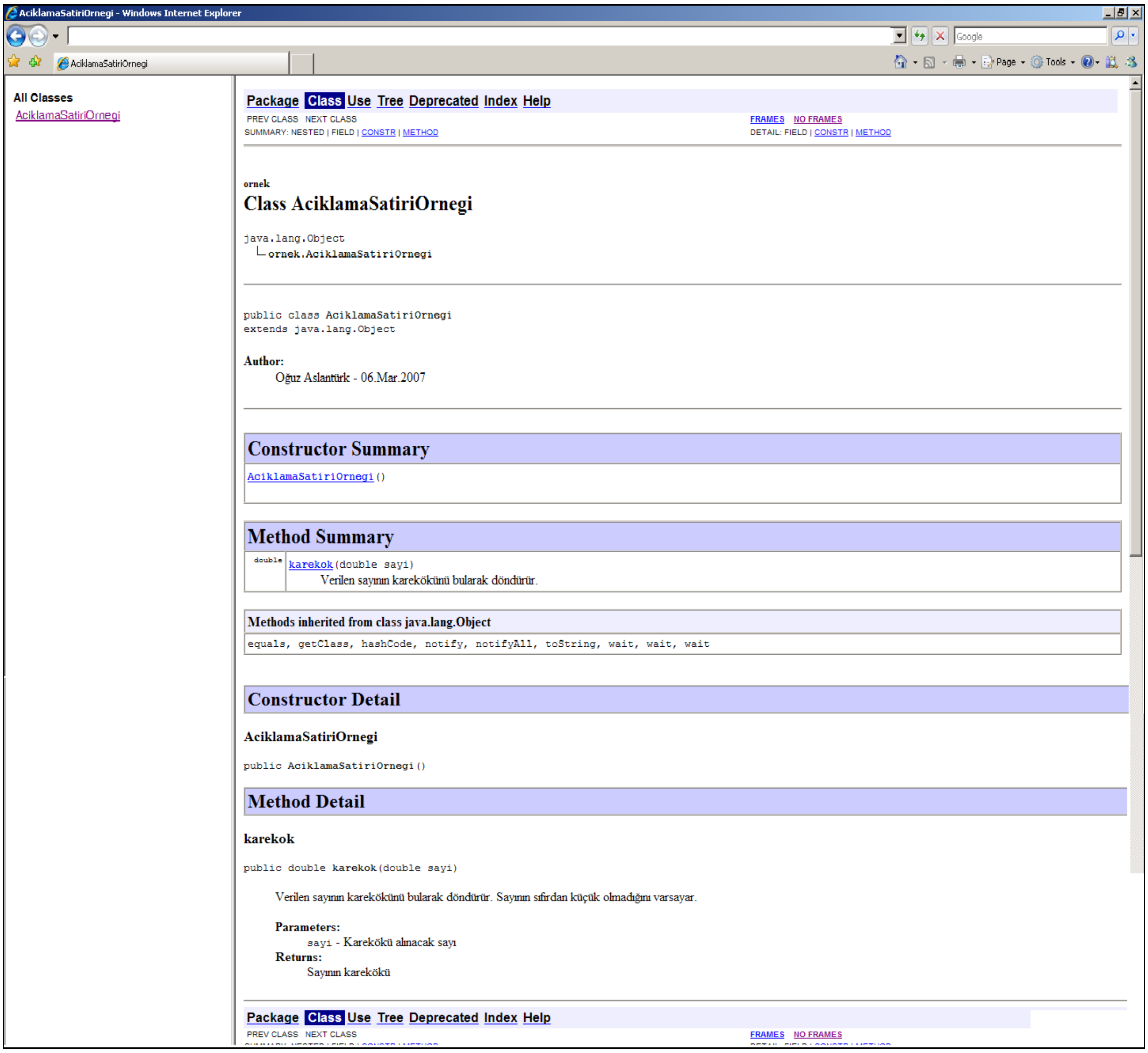
Kod 3-3. javadoc ile yapılan açıklamalar

Konsolda AciklamaSatiriOrnegi.java dosyasının bulunduğu klasöre gidip aşağıdaki satırı yazarsak, Docs adlı bir klasör oluşturulacak ve adı verilen .java dosyası için HTML belgeleri oluşturulacaktır.

```
javadoc -d Docs AciklamaSatiriOrnegi.java
```

Kod 3-4. Javadoc ile belge oluşturma

Aşağıda bu belgeye ait bir görünüm bulunmaktadır:


AciklamaSatiriOrnegi - Windows Internet Explorer

All Classes
AciklamaSatiriOrnegi

Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

ornek
Class AciklamaSatiriOrnegi

```
java.lang.Object
  └ ornek.AciklamaSatiriOrnegi
```

```
public class AciklamaSatiriOrnegi
extends java.lang.Object
```

Author:
Oğuz Aslantürk - 06.Mar.2007

Constructor Summary
AciklamaSatiriOrnegi()

Method Summary
double karekok(double sayi)
Verilen sayının karekökünü bularak döndürür.

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

AciklamaSatiriOrnegi

public AciklamaSatiriOrnegi()

Method Detail

karekok

public double karekok(double sayi)

Verilen sayının karekökünü bularak döndürür. Sayının sıfırdan küçük olmadığını varsayar.

Parameters:
sayi - Karekökü alınacak sayı
Returns:
Sayının karekökü

Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES

Kod 3-5. Açıklama satırlarından oluşturulan javadoc belgesi

3.3 Temel Türler (*Primitive Types*)

Java'da temel türler tabloda görüldüğü kadardır. Herbir temel türün uzunluğunun ve alabileceği en büyük ve en küçük değerlerin belli olması programcıların işini kolaylaştırmaktadır.

Tür	İkil sayısı	En küçük değer	En büyük değer	Varsayılan değer
byte	8	-2^7	2^7-1	0

short	16	-2^{15}	$2^{15}-1$	0
int	32	-2^{31}	$2^{31}-1$	0
long	64	-2^{63}	$2^{63}-1$	0
float	32	-3.4×10^{38}, 7 hane	3.4×10^{38}, 7 hane	0.0
double	64	-1.7×10^{308}, 15 hane	1.7×10^{308}, 15 hane	0.0
char	16	-	-	Boşluk krakteri
boolean	0	false	true	false
Nesne referansı	32	-	-	null

Tablo 3-2. Java'da Temel Türler

Temel türlerin varsayılan değerleri görüldüğü gibi belirlidir. Yine de değişkenler tanımlanırken ilk değer olarak bu değerlerin atanması kodun okunurluğunu arttıran ve olası hataların önüne geçen iyi bir programlama alışkanlığıdır.

3.4 Değişken Tanımlama ve İlk Değer Atama (*Variable Declaration and Initialization*)

Değişken, veri saklamak için kullanılan bellek alanlarına verilen addır. Her değişkenin adı, adresi, türü ve değeri vardır. Değişken tanımı aşağıdaki şekilde yapılır:

```
değişkenin_türü değişkenin_adı;
```

Aynı türden birden fazla değişken tanımlanacaksa, bunlar "," işareti ile ayrılarak yazılabilirler.

```
değişkenin_türü değişkenin_adı_1, değişkenin_adı_2;
```

Aşağıda bazı değişken tanımları görülmektedir:

```
short sayi;
int sayi1, sayi2;
double gercelSayi1,
 gercelSayi2;
```

Kod 3-6. Değişken tanımlama

Bir değişkene ilk değer atamak için, değişken tanımının hemen yanında atama işleci kullanılabilir:

```
byte byteTipindeDegisken = -8;
char birinciKarakter = ‘a’, ikinciKarakter = ‘z’;
float gercelSayi = 4.35;
```

Kod 3-7. Değişkene ilk değer verme

Java'da bir takım isimlendirme kuralları/gelenekleri (Naming Conventions) bulunmaktadır. Bu kurallara göre değişken adları küçük harfle başlar, birden fazla sözcükten oluşuyorsa, birinci sözcükten sonraki sözcüklerin yalnızca ilk harfleri büyük diğerleri küçük yazılır: ***`byteTipindeDegisken`, `birinciKarakter` değişkenlerinde olduğu gibi.***

Bir değişken tanımlarken, değişken adının özenle seçilmesi ve değişkenin yapacağı işi belirtmesi güzel bir programlama alışkanlığıdır.

3.4.1 Değişkenin Tanım Alanı (*Scope*)

İki küme parantezi ("**{**" ve "**}**") arasında kalan kod kesimine blok denir. "**{**" işareti bir kod bloğu başlatır ve "**}**" işareti başlatılan kod bloğunu bitirir. Herhangi bir değişken, tanımlandığı kod bloğu içinde fiziksel olarak vardır ve o kod bloğu içine yazılan kod kesimlerinden erişilebilirdir. Bu alana değişkenin tanım alanı denir.

Bir blok içinde aynı değişken adı birden fazla kez kullanılamaz.

Örnek:

```
int n = 5;
System.out.println("n: " + n);                // n: 5
{
      int k = 3;
      System.out.println("k: " + k);          // k: 3
      n = 8;
} // k buraya kadar tanımlı, buradan sonra erişilemez
System.out.println("n: " + n);                // n: 8
```

Kod 3-8. Kod bloğu ve tanım alanı örneği

3.5 İşleçler (*Operators*)

3.5.1 Aritmetik işleçler:

İşleç Adı	Gösterimi	Örnek
Atama (assignment)	=	a = 5;
Toplama	+	a = a + b;
Çıkarma	-	a = a – b;
Çarpma	*	a = a * b;
Bölme	/	a = a / b;
Mod Alma	%	a = a % 3;

Tablo 3-3. Aritmetik işleçler

3.5.2 Birleşik Aritmetik İşleçler:

İşleç Adı	Gösterimi	Örnek
Toplama ve atama	+=	a += b; // a = a + b;
Çıkarma ve atama	-=	a -= b; // a = a - b;
Çarpma ve atama	*=	a *= b; // a = a * b;
Bölme ve atama	/=	a /= b; // a = a / b;
Mod alma ve atama	%=	a %= b; // a = a % b;

Tablo 3-4. Birleşik Aritmetik İşleçler

3.5.3 Arttırma ve Azaltma İşleçleri:

İşleç	Gösterimi	Örnek
Arttırma (increment)	++	a++; // post-increment ++a; // pre-increment
Azaltma (decrement)	--	a--; // post-decrement --a; // pre-decrement

Tablo 3-5. Arttırma ve Azaltma İşleçleri

Arttırma ve azaltma işleçleri değişkenin önüne veya sonuna yazılabilir. Nereye yazıldıklarına bağlı olarak işleçlerin nasıl işletileceği değişir:

Değişkenin sonuna yazılırsa:

```
int a = 3, b = 5, c;
c = a++ + b;
System.out.println("a: " + a);      // a: 4
System.out.println("b: " + b);      // b: 5
System.out.println("c: " + c);      // c: 8
```

Kod 3-9 Arttırma işlecinin değişkenin sonuna yazılması örneği

Değişkenin önüne yazılırsa:

```
int a = 3, b = 5, c;
c = ++a + b;
System.out.println("a: " + a);      // a: 4
System.out.println("b: " + b);      // b: 5
System.out.println("c: " + c);      // c: 9
```

Kod 3-10 Arttırma işlecinin değişkenin önüne yazılması örneği

Örneklerde de görüldüğü gibi, her iki kod kesiminde de `a`'nın değeri arttırma işleci ile 1 artmaktadır. Ancak ilk örnekte, `a`'nın değeri arttırılmadan önce (`a`'nın değeri 3 iken) aktarma işlecinin sağ tarafındaki işlem gerçekleştirilmekte ve buna bağlı olarak `c`'nin değeri 8 olmakta; ikinci örnekte ise `a`'nın değeri 1 arttırıldıktan sonra sağ taraftaki işlem gerçekleştirilmekte ve buna bağlı olarak `c`'nin değeri 9 olmaktadır. Başka bir deyişle, her durumda `a`'nın değeri arttırma işleci ile 1 arttırılacaktır ancak bunun önce mi sonra mı olacağına göre `c`'nin değeri değişmektedir.

> ***Eğer bu işletim kafanızı karıştırdıysa, ++a ya da a++ yerine bir önceki ya da bir sonraki satıra a += 1 yazarak kodun nasıl çalıştığından emin olabilirsiniz. Programlama dilindeki her ayrıntıyı kullanmanız gerekmez, önemli olan doğru, etkin ve okunur kod yazmaktır.***

3.5.4 İlişkisel (*relational*) işleçler:

İşleç Adı	Gösterimi	Örnek
Eşittir	==	a == b
Eşit değildir	!=	a != b
Büyüktür	>	a > b
Küçüktür	<	a < b
Büyük ya da eşittir	>=	a >= b
Küçük ya da eşittir	<=	a <= b

Tablo 3-6. İlişkisel İşleçler

İlişkisel işleçler, mantıksal ifadelerde kullanılır. Bir karşılaştırma sonucu elde edilen değer mantıksal doğru (`true`) ya da mantıksal yanlış (`false`) olabilir.

> ***C ve C++ dillerinde, 0 (sıfır)'dan farklı değerler mantıksal doğru, 0 değeri ise mantıksal yanlış olarak değerlendirilir. Ancak Java'da mantıksal doğru ve yanlışı ifade etmek üzere `boolean` temel türü bulunmaktadır ve herhangi bir sayısal değer doğru ya da yanlış olarak değerlendirilemez. Bu nedenle, sayısal değerlere bağlı olarak kurulan ilişkisel ifadelerde ilişkisel işleç kullanılması zorunludur; C ya da C++'da olduğu gibi sadece değişkenin değerine göre karar verilemez.***

> ***İlişkisel işleciler ile ilgili olarak en yaygın yapılan hata, eşittir işleci ("==") yerine yanlışlıkla atama işleci ("=") yazmaktır. Kodlarınızda buna dikkat ediniz.***

3.5.5 Mantıksal İşleçler:

İşleç adı	Gösterimi	Örnek
ve (*and*)	&&	a > 0 && a < 10
ya da (*or*)	||	a < 0 || a > 10
değil (*not*)	!	!(a > 0)

Tablo 3-7. Mantıksal İşleçler

3.5.6 İşleçlerin Öncelikleri:

Verdiğimiz işleçlerle ilgili öncelik kurallarını basit ifadelerle açıklayalım.

- Tekil işleçlerin ikili işleçlere göre önceliği vardır.
- Aritmetik işleçlerde çarpma ve bölme işleçlerinin toplama ve çıkarma işleçlerine göre önceliği vardır. Aynı öncelikli işleçler soldan sağa sırayla işletilir.
- Aritmetik işleçler ilişkisel işleçlere göre önceliklidir.
- İlişkisel işleçler soldan sağa sırayla işletilir.
- İlişkisel işleçler mantıksal işleçlere göre önceliklidir.
- En düşük öncelikli işleç atama işlecidir.

Yukarıdaki kurallara göre aşağıdaki kod kesimini işletelim:

```
int a = 3, b = 5, c;
c = a * ++b / 5 - 12 * a + b;
System.out.println("c: " + c);            // c: -27

boolean bool = a + b < a * c && a + b >= b + a;
System.out.println("bool: " + bool);      // bool: false
```

Kod 3-11. İşleç öncelikleri örneği

Öncelikler parantez kullanılarak değiştirilebilir. Bir ifade, bazı kısımları parantezler arasına alınarak düzenlendiğinde önce parantezlerin içi işletilir. Kod 3-10’da görülen kod kesimini, öncelikleri parantezlerle belirleyecek şekilde yeniden yazıp işletelim:

```
int a = 3, b = 5, c;
c = (a * ++b) / (5 - 12) * (a + b);
System.out.println("c: " + c);            // c: -18

boolean bool = (a + b) < (a * c) && (a + b) >= (b + a);
System.out.println("bool: " + bool);      // bool: false
```

Kod 3-12. Parantezler ile öncelik belirleme örneği

Görüldüğü gibi `c` değişkeninin değerinin hesaplandığı satırda öncelikleri değiştirerek farklı bir değer bulunmasını sağladık.

`bool` değişkeninin değerinin belirlendiği satırda ise öncelikleri değiştirmedik. Ancak kodun daha kolay okunabilir olmasını sağladık.

Öncelik kuralları bilinse de karmaşık bir deyimin okunabilirliği çoğu zaman oldukça düşüktür. Parantez, öncelikleri değiştirmek üzere kullanılabildiği gibi, bazen sırf kodun okunabilirliğini arttırmak için kullanılır.

Kodun okunurluğunu arttırmak için parantezleri kullanmak güzel bir programlama alışkanlığıdır.

Bu bölümde listelenen işleçler, Java dilindeki bütün işleçler değildir. Ancak vereceğimiz temel bilgiler açısından bu işleçler yeterlidir. İşleçlerle ilgili daha ayrıntılı bilgi için Java referans belgelerine başvurabilirsiniz.

3.6 Denetim Deyimleri (*Control Statements*)

Denetim deyimleri, bir takım koşullara göre programın akışının belirlenmesini sağlayan programlama ögeleridir. Bir denetim deyimi, işletilecek bir sonraki koşulun hangisi olacağını belirlemek üzere bir koşulu denetler.

3.6.1 `if – else` deyimi

Şu şekilde yazılır:

```
if (koşul ifadesi) {
      koşulun doğru olması durumunda işletilecek kod kesimi
}
else {
      koşulun doğru olmaması durumunda işletilecek kod kesimi
}
```

`if` ve **`else`** deyimleri ayrılmış sözcüklerdir. **`if`** deyiminin denetlemesi istenen koşul ifadesi parantez içinde yazılmak zorundadır. Koşul ifadesini izleyen küme parantezi (**`{`**), **`if`** bloğunun başlangıcıdır. Bu parantez kapatılana (**`}`**) kadarki kod kesimi, koşulun doğru olması durumunda işletilecek kesimdir. **`if`** bloğu kapatıldıktan hemen sonra **`else`** deyimi gelir ve **`else`** bloğu başlar. **`else`** bloğu, koşulun doğru olmaması durumunda

işletilir. Kısaca, koşula bağlı olarak ya **`if`** bloğu ya da **`else`** bloğu işletilecektir. **`if`** ve **`else`** bloklarının arasına herhangi bir deyim giremez.

Örnek:

```
int a = 3, b = 5;
System.out.println("Denetimden önceki kod kesimi");
if (a > b) {
      System.out.println("a, b'den büyüktür.");
}
else {
      System.out.println("a, b'den büyük değildir.");
}
System.out.println("Denetimden sonraki kod kesimi");
```

Kod 3-13. if deyimi örneği - 1

Çıktı:

```
Denetimden önceki kod kesimi
a, b'den büyük değildir.
Denetimden sonraki kod kesimi
```

`else` bloğunun yazılması zorunlu değildir, ancak **`if`** bloğu olmadan **`else`** bloğu yazılamaz. Başka bir deyişle, **`else`**'siz **`if`** olabilir ancak tersi doğru değildir. Bazen yalnızca koşulun doğru olması durumunda işletmek istediğimiz bir kod kesimi vardır ancak yanlış olması durumunda işletilecek özel bir kod yoktur. Bu durumda yalnızca **`if`** bloğunu yazmak yeterlidir.

Örnek:

```
int a = 3, b = 5;
System.out.println("Denetimden önceki kod kesimi");
if (a > b) {
      System.out.println("a, b'den büyüktür.");
}
System.out.println("Denetimden sonraki kod kesimi");
```

Kod 3-14. if bloğu örneği - 2

Çıktı:

```
Denetimden önceki kod kesimi
Denetimden sonraki kod kesimi
```

Eğer **`if`** ya da **`else`** bloklarından herhangi birisi tek satırdan oluşuyorsa, **`{ }`** arasına alınmayabilir. **`if`** deyiminden hemen sonra gelen ilk satır **`if`** deyimine, **`else`** deyiminden sonra gelen ilk satır da aynı şekilde **`else`** deyimine bağlı olarak işletilir.

Örnek:

```
int a = 3, b = 5;
System.out.println("Denetimden önceki kod kesimi");
if (a > b)
      System.out.println("a, b'den büyüktür.");
else
      System.out.println("a, b'den büyük değildir.");
System.out.println("Denetimden sonraki kod kesimi");
```

Kod 3-15. if bloğu örneği - 3

> ***Gerek kodun okunurluğunun kaybolmaması, gerekse dalgınlıkla yapılabilecek hataların önüne geçilebilmesi açısından, `if` ya da `else` bloklarını tek satırdan oluşsalar bile { } arasına almak iyi bir programlama alışkanlığıdır.***

Bir **`if`** bloğunun içine başka **`if`** deyimi ya da deyimleri de gelebilir. Burada herhangi bir sınır yoktur. Algoritmaya göre, içiçe **`if`** deyimleri yazılabilir.

Örnek:

```
int a = 7, b = 3, c = 1;
if (a > b) {
      System.out.println("a, b'den büyüktür.");
      if (a > c) {
            System.out.println("a, c'den de büyüktür.");
      }
      else {
            System.out.println("a, c'den büyük değildir.");
      }
}
```

Kod 3-16. İçiçe if deyimleri

Çıktı:

```
a, b'den büyüktür.
a, c'den de büyüktür.
```

3.6.2 **`if – else if`** deyimi

Şu şekilde yazılır:

```
if (1. koşul ifadesi) {
      1. koşulun doğru olması durumunda işletilecek kod kesimi
}
else if (2. koşul ifadesi) {
      2. koşulun doğru olmaması durumunda işletilecek kod kesimi
}
... // istenildiği kadar else if bloğu
else {
      hiçbir koşulun doğru olmaması durumunda işletilecek kod kesimi
}
```

`if - else if` deyimleri, bir koşulun sağlanmadığı durumda diğer koşul ya da koşulların denetlenmesini sağlayan bir yapıdadır. Denetlenen koşullardan herhangi birisinin sonucu mantıksal doğru olursa, o kod bloğu işletilir ve programın akışı en sonraki **`else`** bloğunun bitimine sapar. Hiç bir koşul doğru değilse, o zaman **`else`** bloğu işletilir.

`else` bloğunun yazılması zorunlu değildir.

Örnek:

```
int a = -1;
System.out.println("Denetimden önceki kod kesimi");
if (a > 0) {
      System.out.println("a, 0'dan büyüktür.");
}
else if (a < 0) {
      System.out.println("a, 0'dan küçüktür.");
}
else {
      System.out.println("a'nın değeri 0'dır.");
}
System.out.println("Denetimden sonraki kod kesimi");
```

Kod 3-17. if – else if deyimi örneği

Çıktı:

```
Denetimden önceki kod kesimi
a, 0'dan küçüktür.
Denetimden sonraki kod kesimi
```

Çıktıda görüldüğü gibi, kod bloklarından yalnızca birisi işletilmiştir.

3.6.3 `switch` deyimi

`switch` deyimi, **`if-else-if`** deyimleri ile yazılabilecek bir kod kesimini daha basitçe yazmayı sağlayan, bir ifadenin değerine göre dallanmayı sağlayabilen bir deyimdir.

Aşağıdaki gibi yazılır:

```
switch (ifade) {
      case deger1:
            ifadeler;
            break;
      case deger2:
            ifadeler;
            break;
      ....
      case degerN:
            ifadeler;
            break;
      default:
            ifadeler;
}
```

`switch` deyiminden sonra parantez içerisine bir ifade yazılır. Bu ifade, tek başına bir değişken olabileceği gibi, herhangi bir matematiksel işlem ya da işlev çağrısı da olabilir. **`case`** satırları, ifadenin değerini denetler. Herhangi bir **`case`** satırında eşitlik yakalanabilirse, o **`case`** satırı işletilir. **`break`** deyimi, denetim yapısından çıkmaya yarayan deyimdir ve bir **`case`** satırı işletildikten sonra diğerlerinin işletilmemesini sağlar. Ancak yazılması zorunlu değildir. **`default`** ise, hiçbir **`case`** satırında ifadenin değerinin yakalanamaması durumunda işletilmek istenen kod kesiminin kodlanacağı yerdir, yazılması zorunlu değildir.

```
int ay = 5;
```

```
System.out.println("switch'ten önceki kod kesimi");
switch (ay) {
      case 1: System.out.println("Ocak");       break;
      case 2: System.out.println("Şubat");      break;
      case 3: System.out.println("Mart");       break;
      case 4: System.out.println("Nisan");      break;
      case 5: System.out.println("Mayıs");      break;
      case 6: System.out.println("Haziran");    break;
      case 7: System.out.println("Temmuz");     break;
      case 8: System.out.println("Ağustos");    break;
      case 9: System.out.println("Eylül");      break;
      case 10: System.out.println("Ekim");      break;
      case 11: System.out.println("Kasım");     break;
      case 12: System.out.println("Aralık");    break;
      default: System.out.println("Ay 1 - 12 aralığında değil");
}
System.out.println("switch'ten sonraki kod kesimi");
```

Kod 3-18. switch deyimi örneği

Çıktı:

```
switch'ten önceki kod kesimi
Mayıs
switch'ten sonraki kod kesimi
```

***switch* deyimi kullanılırken *break* deyimlerinin dikkatli kullanılması gerekir. *break* deyiminin unutulması ya da yazılmaması durumunda, *case* satırı işletildikten sonra *switch* deyiminin sonuna sapılmayacak, işleyiş bir sonraki *case* satırından devam edecektir. *if-else if* kod blokları, *switch* bloklarına oranla hata yapmanıza daha az yatkındır.**

Örnek:

```
int ay = 5;
switch (ay) {
      case 12:
      case 1:
      case 2: System.out.println("Kış"); break;
      case 3:
      case 4:
```

```
      case 5: System.out.println("İlkbahar");  break;
      case 6:
      case 7:
      case 8: System.out.println("Yaz"); break;
      case 9:
      case 10:
      case 11: System.out.println("Sonbahar"); break;
      default: System.out.println("Ay 1 - 12 aralığında değil");
}
```

Kod 3-19. break kullanılmayan switch deyimi örneği

Çıktı:

```
İlkbahar
```

3.7 Döngüler (Loops)

Döngüler, belirli bir koşul sağlandığı sürece tekrarlanması gereken işler için kullanılan programlama ögeleridir. Java'da **`for`**, **`while`** ve **`do-while`** olmak üzere 3 adet döngü deyimi bulunmaktadır.

Herhangi bir algoritmada hangi döngünün kullanılması gerektiği ile ilgili kesin kurallar yoktur. Döngüler uygun değişiklikler ile aynı işi yapacak şekilde kodlanabilirler. Ancak genel eğilim, kaç kere döneceği belli olmayan, koşulun bir girdiye (örneğin bir dosyadan okunan ya da kullanıcının girdiği değerlere) göre denetlendiği durumlarda **`while`** ya da **`do-while`** döngüsünü, diziler gibi, tekrar sayısının belli olduğu durumlarda **`for`** döngüsünü kullanmak yönündedir.

3.7.1 `while` döngüsü

Şu şekilde yazılır:

```
while (koşul ifadesi) {
      deyimler;
}
```

`while` sözcüğü ile koşulun içine yazıldığı parantezler zorunludur. Koşul sağlandığı sürece **`while`** bloğunun içindeki kod kesimi işletilecektir.

Örnek:

```
int i = 1;
while (i <= 5) {
      System.out.println(i + ". kez işletildi..");
      i++;
}
```

Kod 3-20. while döngüsü örneği

Çıktı:

```
1. kez işletildi..
2. kez işletildi..
3. kez işletildi..
4. kez işletildi..
5. kez işletildi..
```

3.7.2 `do - while` döngüsü

Şu şekilde yazılır:

```
do {
      deyimler;
} while (koşul ifadesi);
```

`do` ve **`while`** sözcükleri ile koşulun için yazıldığı parantezler zorunludur. Koşul sağlandığı sürece **`do-while`** bloğunun içindeki kod kesimi işletilecektir.

Örnek:

```
int i = 1;
do {
      System.out.println(i + ". kez işletildi..");
      i++;
} while (i <= 5);
```

Kod 3-21. do-while döngüsü örneği

Çıktı:

```
1. kez işletildi..
2. kez işletildi..
3. kez işletildi..
4. kez işletildi..
5. kez işletildi..
```

`do-while` döngüsünde, döngü bloğu içindeki kod kesimi en az bir kez mutlaka işletilecektir. Çünkü önce döngü bloğu işletilip sonra koşul denetlenmektedir. `while` döngüsünde ise önce koşula bakılıp sonra döngü bloğu işletildiği için, döngüye hiç girilmemesi olasıdır.

3.7.3 `for` döngüsü

Şu şekilde yazılır:

```
for (ilk_deger_atama; koşul; her_dönüşte) {
      deyimler;
}
```

Bu yazılışta **`for`** sözcüğü, ***`ilk_deger_atama`***, ***`koşul`*** ve ***`her_dönüşte`*** alanlarının içine yazıldığı parantezler ve ";" (noktalı virgül) işaretlerinin yazılması zorunludur.

`ilk_deger_atama`: Döngüye ilk defa girilirken işletilecek kod kesimidir. Yalnızca bir kez işletilir. Boş bırakılabilir ya da birden fazla deyimden oluşabilir. Birden fazla deyimden oluşuyorsa, deyimler "," (virgül) işareti ile birbirlerinden ayrılır. Genellikle döngü değişkeninin ilk değerinin verilmesi için kullanılır ve tek deyim içerir.

`koşul`: Döngü bloğu işletilmeden önce denetlenecek koşul ifadesini içerir. **`while`** döngüsünde olduğu gibi, önce koşul denetlenir, koşulun mantıksal doğru olması durumunda döngü bloğu işletilir.

`her_dönüşte`: Döngü bloğu işletildikten sonra, koşul yeniden denetlenmeden hemen önce işletilecek kod kesimidir. Döngü bloğunun her işletilmesinden sonra bir kez işletilir. Boş bırakılabilir ya birden fazla deyimden oluşabilir. Birden fazla deyimden oluşuyorsa, deyimler "," (virgül) işareti ile birbirlerinden ayrılır. Genellikle döngü değişkeninin değerinin güncellenmesi için kullanılır ve tek deyim içerir.

Örnek:

```
int i;
for (i = 1; i <= 5; i++) {
      System.out.println(i + ". kez işletildi..");
```

```
}
```

Kod 3-22. for döngüsü örneği - 1

Çıktı:

```
1. kez işletildi..
2. kez işletildi..
3. kez işletildi..
4. kez işletildi..
5. kez işletildi..
```

Aynı döngü şu şekilde de yazılabilir:

```
for (int i = 1; i <= 5; i++) {
      System.out.println(i + ". kez işletildi..");
}
```

Kod 3-23. for döngüsü örneği – 2

Görüldüğü gibi döngü değişkeni ***`ilk_deger_atama`*** kısmında da tanımlanabilir. Böyle tanımlanması durumunda, döngü değişkeninin (bu kodda **`i`** adlı değişken) tanım alanı **`for`** döngüsü ile sınırlıdır. Başka bir deyişle **`for`** döngüsünü bitiren **`}`** işaretinden sonra döngü değişkeni tanımsızdır.

Aynı döngü ***`ilk_deger_atama`*** ve ***`her_dönüşte`*** kısımları boş bırakılacak şekilde yazılabilir:

```
int i = 0;
for (  ; i <= 5;  ) {
      System.out.println(i + ". kez işletildi..");
      i++;
}
```

Kod 3-24. for döngüsü örneği - 3

> ***`for` döngüsünü yalnızca koşul alanını kullanıp diğer alanları kullanmadan kodladığımızda, `while` döngüsüne çok benzediği görülüyor. Bu durumda `for` yerine `while` kullanmak daha anlamlı olabilir!***

`ilk_deger_atama` ve ***`her_dönüşte`*** kısımlarında virgül kullanımını şu şekilde örnekleyebiliriz:

```
for (int i = 1, j = 10; i <= j; i++, j--) {
```

```
      System.out.println("i: " + i + ", j: " + j);
}
```

Kod 3-25. for döngüsü örneği - 4

Çıktı:

```
i: 1, j: 10
i: 2, j: 9
i: 3, j: 8
i: 4, j: 7
i: 5, j: 6
```

Aynı döngü şu şekilde de yazılabilir:

```
int j = 10;
for (int i = 1; i <= j; i++) {
      System.out.println("i: " + i + ", j: " + j);
      j--;
}
```

Kod 3-26. for döngüsü örneği - 5

3.8 Özet

Java, C, C++ ve C# dillerine çok benzer. Sözdizim kuralları, değişken tanımlamaları ve ilk değer atama kuralları, denetim ve döngü yapıları neredeyse aynıdır.

Java'da açıklama satırları **`javadoc`** etiketleri ve aracı ile diğerlerinden ayrılmaktadır. Benzer araçlar ise Java teknolojilerinin çoğunda kod ya da belge üretmek amacıyla sıklıkla kullanılmaktadır.

4 Uygulama Geliştirme Ortamı

Java ile uygulama geliştirmek için çeşitli Tümleşik Uygulama Geliştirme Ortamı (*Integrated Development Environment - IDE*) yazılımları kullanılmaktadır. Ancak bu ortamların birçok ayrıntıyı kullanıcıdan gizleyerek çalışıyor olmaları, platformu ve dili yeni öğrenen kişilerin gerçekte neler olup bittiğini bilmeden kod yazabilmeleri sonucunu doğurmaktadır. Tamamen yabancı bir ortamda fazla birşey bilmeden çalışır programlar oluşturabilmek başlangıçta cazip gelse de, öğrenme eylemi açısından göründüğü kadar faydalı değildir. Amaç öğrenmek olduğuna göre önce böyle bir araç olmadan işlerin nasıl yapılacağını görmek daha doğru olacaktır. Sonraki aşamada ise daha hızlı ve kolay çalışabilmek için elbette uygulama geliştirme ortamlarından faydalanmak gerekir.

4.1 JDK ve JRE

JDK, *Java Development Kit* sözcüklerinin baş harflerinden oluşturulmuş bir kısaltmadır. Java programlarının geliştirilip çalıştırılabilmesi için gerekli araç ve kütüphaneleri içeren paket JDK adı ile sunulmaktadır.

JRE ise *Java Runtime Environment* sözcüklerinin baş harflerinden oluşur. Java programlarını çalıştırmak (geliştirmek için değil!) için gerekli bileşenleri içerir.

Eğer bilgisayarınızda Java ile program geliştirmeyecek, yalnızca Java programlarını çalıştıracaksanız, bilgisayarınızda JRE bulunması yeterlidir. Ancak eğer kod geliştirecekseniz, JDK'ya gereksinim duyarsınız.

JDK kurulum paketinin içinde JRE de gelmektedir. Yani ikisini ayrı ayrı indirip kurmanıza gerek yoktur.

http://java.sun.com/javase/downloads/index.jsp adresinden JDK'yı indirmek için gerekli bağlantıya tıklayın. Bu bağlantı sizi farklı platformlara ait JDK'ları indirebileceğiniz sayfaya götürecektir. Örneklerimizi WindowsXP üzerinde yazacağımız için biz `jdk-6-windows-i586.exe`

(kitabın yazılma zamanında JDK'nın bu sürümü var) dosyasını indirdik. Ancak siz kendi kodlarınızı hangi platformda yazacaksınız, ona uygun JDK'yı seçebilirsiniz.

JDK kurulum dosyasına çift tıkladığınızda iki ayrı kurulum başlatılacaktır. Bunlardan ilki JDK'nın kurulumudur. Varsayılan kurulumu gerçekleştirebilir ya da kendinize göre özelleştirme yaparak hedef klasörü değiştirebilir, demoları ve kaynak kodları kurmamayı seçebilirsiniz.

İkinci kurulum ise *Public JRE* adlı bir programa aittir. Bu kurulumu varsayılan haliyle gerçekleştiriniz. *Public JRE,* makinanızda JVM ortamını oluşturur ve JVM gereksinimi olan diğer programlarınıza da bu ortamdan faydalanma olanağı sağlar.

JDK'yı makinanıza kurduktan sonra, öncelikle çevre değişkenlerini güncellemeniz gerekir:

`PATH değişkeni`: Güncellenmesi gereken çevre değişkenlerinden ilki **`PATH`** değişkenidir. JDK'yı kurduğunuz klasörün altındaki **`bin`** klasörünü **`PATH`** değişkenine tanımlamalısınız. Bunu WindowsXP'de şu şekilde yapabilirsiniz:

- Başlat (*Start*) -> Ayarlar (*Settings*) -> Denetim Masası (*Control Panel*) -> Sistem (*System*) penceresini açın.
- Gelişmiş (*Advanced*) sekmesine gelin.
- Çevre Değişkenleri (*Environment Variables*) tuşuna tıklayın.
- Alttaki Sistem Değişkenleri (*System variables*) penceresinde **`PATH`** adlı değişkeni bulup Düzenle (*Edit*) tuşuna tıklayın.
- Bu değişkenin içinde tanımlı değerler varsa bunları değiştirmeyin, en sona gidip ";" yazdıktan sonra JDK'yı kurduğunuz klasörün adını buraya ekleyin.

`PATH` değişkenine bir klasör eklendiği zaman, eklenen klasörün altındaki dosyalar bilgisayarın herhangi bir klasöründen erişilebilir hale gelir. Böylece bir çalışma klasöründe çalışırken JDK'nın **`bin`** klasörünün altında

bulunan araçları rahatça kullanabilirsiniz. Bu araçlardan bazılarına hızlıca bakalım:

- **`javac.exe`**: Java programlarını derlemek için kullanılır.
- **`java.exe`**: Derlenmiş Java programlarını çalıştırmak için kullanılır.
- **`javadoc.exe`**: Java kaynak kodlarındaki kod içi belgelemeyi kullanarak belge oluşturmak için kullanılır.
- **`appletviewer.exe`**: Applet kodlarını çalıştırmak için kullanılır.
- **`javap.exe`**: Derlenmiş bir sınıf dosyasının içinde neler olduğunu görmek için kullanılır.
- **`jar.exe`**: Java programlarını oluşturan dosyaları sıkıştırarak paketlemek ya da bu paketleri açmak için kullanılır. **`zip`** biçiminde (*format*) dosyalar üretebilir.

`CLASSPATH değişkeni`: Güncellenmesi gereken bir diğer çevre değişkeni ise **`CLASSPATH`** değişkenidir. Eğer çevre değişkenleri arasında **`CLASSPATH`** değişkeni tanımlıysa bunu ya silin ya da içeriğini yalnızca "." olacak şekilde güncelleyin.

> ***Bazı programlar `PATH` değişkenine kendi kurulumları ile birlikte gelen JDK veya JRE'yi eklerler. Bu durumda `PATH` değişkeninin içinde farklı JDK sürümlerine ait olan ve ";" işaretiyle ayrılmış birden çok tanımlama bulunabilir. Bu ise Java programlarını derleyememenize ya da çalıştıramamanıza neden olabilir. Eğer `PATH` değişkeninde öyle tanımlar varsa silin.***

4.2 Java ile ilk program

Bu kısımda yazılan programlarda henüz anlatmadığım bazı deyimler ve anahtar sözcükler kullanacağım. Bunları kullanmamın nedeni zorunlu olmam. Çünkü Java ile programlama yaparken uymamız gereken bazı kurallar var. Ancak henüz bu kurallardan bahsetmediğim için, yazdığım programların bu kısımlarını açıklamayacak, şimdilik bunları birer

programlama kalıbı olarak öğrenmenizi isteyeceğim. Daha sonra ilgili konular açıklanırken bu konulara geri dönecek ve gerekli açıklamaları yapacağım.

Bir programlama dili öğrenirken, öncelikle kendinize bir çalışma alanı oluşturmakta fayda olabilir. Örneğin `C:\calisma` gibi bir klasör oluşturup yazdığınız bütün programları bunun altında açacağınız alt klasörlerde saklayabilirsiniz. Ben de buradaki örneklerde `C:\calisma` adlı klasörü kullanacağım.

4.2.1 İlk program

Aşağıda ekrana "İlk Java programımız" yazdıran basit bir programın kaynak kodunu görebilirsiniz.

```
public class IlkProgram {
	public static void main(String[] args) {
		System.out.println("İlk Java programımız..");
	}
}
```

Kod 4-1. İlk Java programı

Bu kaynak kodu herhangi bir metin düzenleyici ile (örneğin Notepad ile) yazıp `IlkProgram.java` adlı bir dosyaya kaydediniz. Dikkat ederseniz, kaynak kodu içine yazdığım dosya ile kodun başındaki sınıf adı (`IlkProgram`) aynı ve büyük-küçük harf duyarlı (*case-sensitive*). Yani dosyanın adını örneğin `ilkprogram.java` yapmamalısınız.

Kural: Java sınıfının adı ile bu sınıfın kaydedildiği dosyanın adı büyük-küçük harf duyarlı olacak şekilde AYNI olmalıdır!

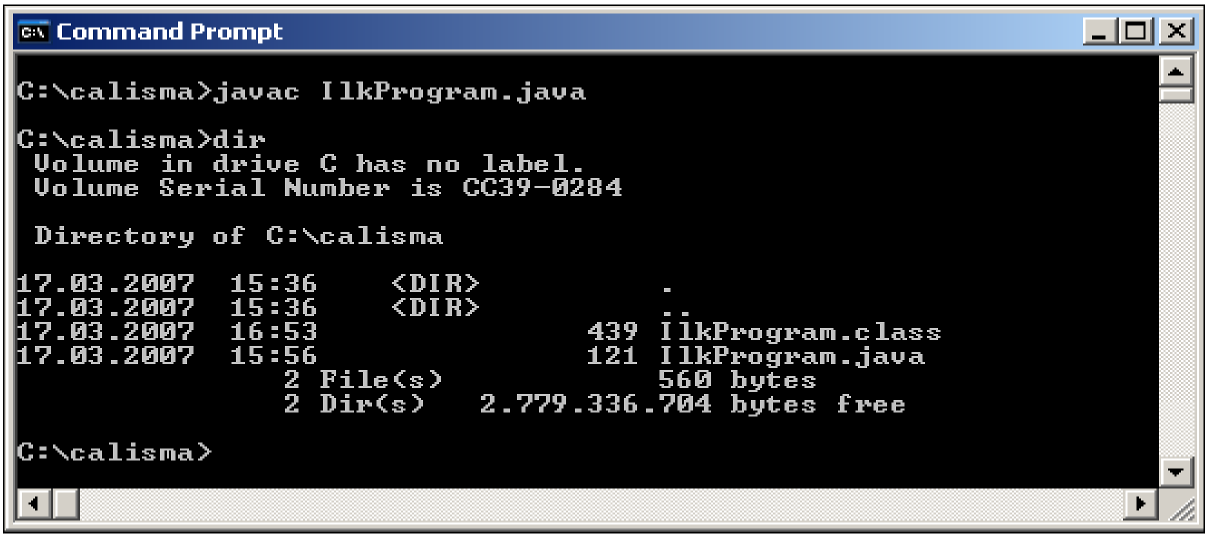
4.2.2 Derleme

`IlkProgram.java` dosyasını **`C:\calisma`** altına kaydettik. Şimdi bu programı derlemek için konsolu açıyoruz.

Konsolu açmak için Başlat (Start) -> Çalıştır (Run) menüsünü seçip buraya `cmd` yazıp `Enter` tuşuna basabilir ya da Başlat (Start) -> Donatılar (Accessories) -> Komut İstemi (Command Propmt) menüsünü seçebilirsiniz.

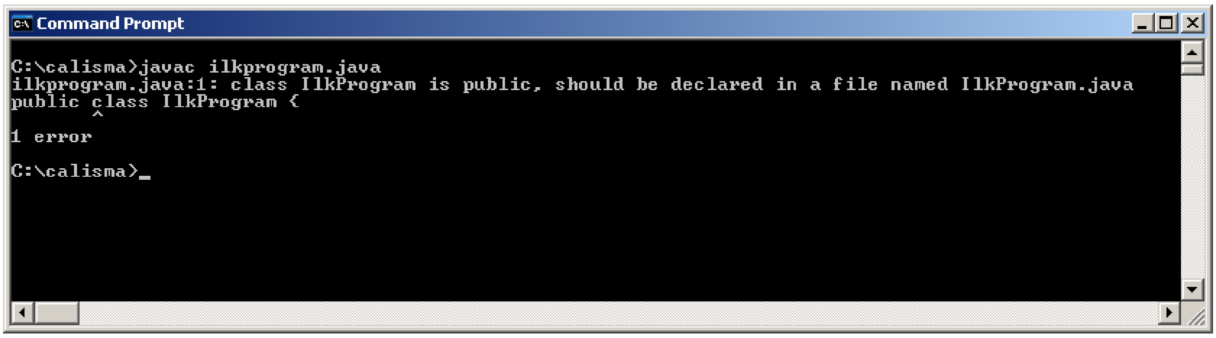
Konsolda `C:\calisma` klasörüne gidiyoruz: **`cd c:\calisma`**

Şimdi programımızı derliyoruz: **`javac IlkProgram.java`**



Şekil 4-1. IlkProgram.java dosyasının derlenmesi

Şekilde görüldüğü gibi program derlendikten sonra **`IlkProgram.class`** adlı, byte-kod içeren Sınıf Dosyası (*Class file*) oluştu. Eğer herhangi bir sözdizim hatası olsaydı derleme işlemi başarısız olacak ve sınıf dosyası oluşmayacaktı. Örneğin, eğer kaynak kodu içeren dosyayı **`ilkprogram.java`** adı ile (sınıf adı ile dosya adı aynı olmayacak şekilde) kaydetseydik bir derleme hatası alacaktık ve sınıf dosyası oluşmayacaktı:



Şekil 4-2. ilkprogram.java dosyasının derlenmesi

4.2.3 Çalıştırma ve `main` yöntemi

Programı başarı ile derlediğimize göre artık çalıştırabiliriz. Ancak çalıştırmadan önce programın nasıl çalışacağına bir bakalım.

Derlenmiş bir sınıf dosyası, eğer içinde

```
public static void main(String[] args) {
....
}
```

şeklinde yazılmış bir yöntem varsa çalıştırılabilir bir dosyadır. Bu dosyayı çalıştırdığımız zaman (az sonra nasıl çalıştıracağımızı göreceğiz), program **`main`** yönteminin ilk satırından çalışmaya başlar.

Örneğimizde, **`main`** yönteminin içinde yalnızca bir satır kod bulunuyor:

```
    System.out.println("İlk Java programımız..")
```

`System.out.println()` yöntemi, "" işaretleri arasında verilen dizgiyi ekrana yazdıran bir yöntemdir. Dolayısıyla programımızı çalıştırdığımızda ekranda

```
    İlk java programımız..
```

yazmasını bekliyoruz.

Bir sınıf dosyasını çalıştırmak için konsolda

`java IlkProgram`

yazıp `Enter` tuşuna basmamız gerekiyor. Dikkat ederseniz,

`java IlkProgram.class`

yazmıyoruz! **`java IlkProgram`** ifadesi ile JVM'ye hangi ***sınıfı*** çalıştırmak istediğimizi belirtiyoruz. JVM, adını verdiğimiz **`IlkProgram`** adlı sınıfı Sınıf Yükleyici görevi ile belleğe yükleyip, bu sınıfın içindeki **`main`** yöntemini işletmeye başlıyor. Burada, **`main`** yönteminin özel bir yöntem olduğunu, çalıştırılmak istenen bir sınıf dosyasında JVM tarafından aranan bir yöntem olduğunu ve bulunabilmesi için de aynen burada yazıldığı şekilde yazılması gerektiğini söyleyelim. Başka bir deyişle, eğer yazdığımız sınıfı çalıştırmak istiyorsak, o sınıfın içinde

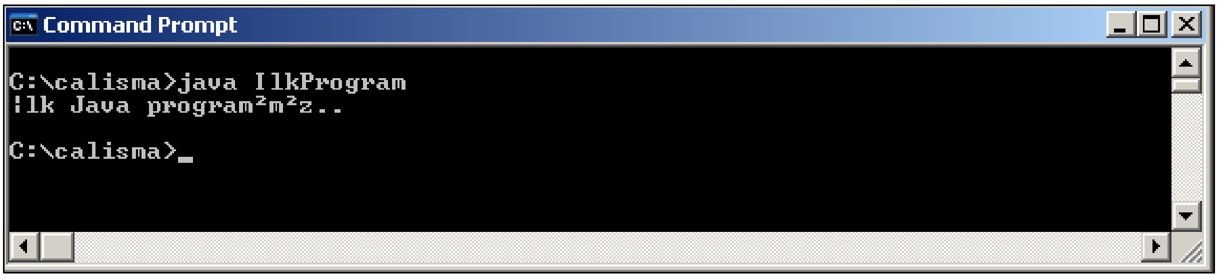
```
    public static void main(String[] args)
```

adlı bir yöntem olması gerekiyor.

`main`* yöntemi yazılırken, *`args`* yerine başka herhangi birşey yazılabilir. Ancak genellikle *`args`* sözcüğü kullanılır ve bu sözcük *`arguments`

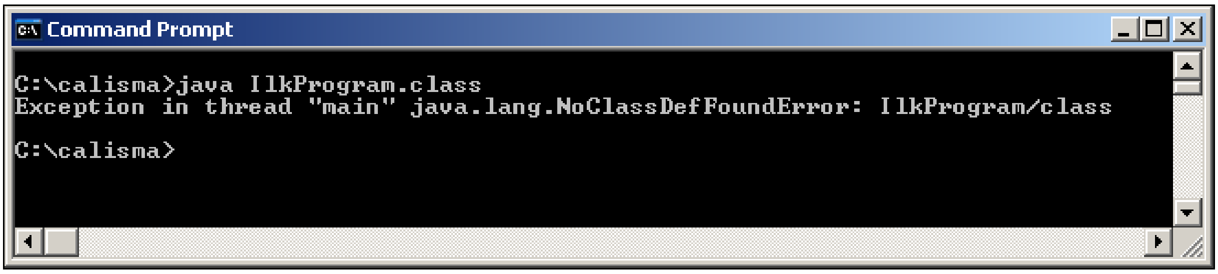
sözcüğünün kısaltmasıdır. `arguments` ile ifade edilen, sınıf çalıştırılırken JVM tarafından bu sınıfa verilen argümanlardır. Bunula ilgili daha ayrıntılı bilgiyi daha sonra vereceğiz.

Aşağıda programın çalıştırılışını görebilirsiniz:



Şekil 4-3. IlkProgram sınıfının çalıştırılması

Eğer **`java IlkProgram.class`** yazarsak ne olacağına da bakalım:



Şekil 4-4. IlkProgram sınıfının hatalı çalıştırılması

Görüldüğü gibi bir hata oluştu ve programımız çalıştırılamadı. Şimdilik programın çalışmadığını görmekle yetinelim ve hatanın ne olduğu ile ilgili ayrıntılı bilgiyi sonraya bırakalım.

4.3 Eclipse Platformu

Java programlarının geliştirilmesi ve çalıştırılması için JDK'nın nasıl kullanılabileceğini gördük. JDK, temelde Java programlarının gerçekleştirilmesi için yeterli olsa da büyük uygulamaların daha hızlı ve kolay geliştirilebilmesi için birçok tümleşik uygulama geliştirme ortamı bulunmaktadır. Bu ortamlar hem kolay kullanılmaları sayesinde, hem de sundukları birçok araç yardımı ile programcının işini kolaylaştırır ve azaltırlar.

Bu kısımda, birçok ücretli ürünün arasından sıyrılmayı başarmış, açık kaynak kodlu ve ücretsiz bir uygulama geliştirme platformu olan Eclipse Platformu'na hızlı bir bakış atılacaktır.

Bir teknoloji, programlama dili.. vb öğrenirken, önce hiçbir araç kullanmadan işlerin nasıl yapılabildiğini öğrenin. Sonra aracı kullanarak bu işleri kolay yapmanın rahatlığını görün. Böylece hem kullandığınız aracın size ne getirdiğini daha kolay farkedersiniz, hem de herhangi bir başka araçla çalışabilecek bilgi birikimine sahip olursunuz. Sonuçta bütün araçlar, o araçlar kullanılmadan da yapılabilecek olan işleri daha kolay yapmanın yollarını sunarlar. Siz neler olup bittiğini bilirseniz hangi araçla çalıştığınızın bir bağlayıcılığı ve önemi kalmaz.

5 Java ile Nesneye Yönelik Programlama

Bu bölümde NYP kavramları üzerinde duracak, bu kavramların Java'da nasıl gerçekleştirildiğine bakacağız.

5.1 Sınıf Tanımları

Sınıf ve nesne kavramlarından daha önce bahsetmiştik. Sınıf, bir grup nesnenin ortak özellik ve davranışlarını içeren tanım, nesne ise bu tanıma göre oluşturulmuş fiziksel bir varlıktı. NYP ise bir nesnenin başka bir nesneye ileti göndermesi, iletiyi alan nesnenin bu iletiye uygun davranışı sergileyerek kendi durumunda bir değişiklik yapması şeklinde gerçekleştiriliyordu (Bkz: Nesneye Yönelik Programlama).

Görüldüğü gibi, nesnelerimizi oluşturabilmek için önce o nesnelerin ait oldukları sınıfları tanımlamamız gerekiyor. Sınıf tanımlarının içine özellikler ve davranışları yazacağız.

Örnek olarak **`Zaman`** adlı bir sınıf tanımı yapalım. **`Zaman`** sınıfımızın **`saat`**, **`dakika`** ve **`saniye`** adlı özellikleri ile **`ilerle()`** adlı bir davranışı olsun. **`saat`**, **`dakika`** ve **`saniye`** özelliklerini, **`Zaman`** sınıfına ait bir nesnenin durumunu oluşturacak tamsayı değerler, **`ilerle()`** davranışını ise bu nesneye **`ilerle`** iletisi gönderildiğinde, o nesneyi bir saniye sonrasına götüren bir davranış olarak düşünelim.

Sınıf tanımı yapılırken, özellikler sınıfın içine değişkenler olarak, davranışlar ise işlevler olarak kodlanır.

> ***Özellik (property) NYP bağlamında, nesnenin durumunu oluşturan, nesnenin ait olduğu sınıfın içine kodlanmış herbir değişkene verilen isimdir. Özellik, Java bağlamında nitelik (attribute) olarak adlandırılır.***
>
> ***Davranış (behaviour) NYP bağlamında, nesnenin durumunda değişiklik yapabilen, nesnenin ait olduğu sınıfın içine kodlanmış herbir işleve verilen isimdir. Davranış, Java bağlamında yöntem (method) olarak adlandırılır.***

Java’da Yöntemler

Java’da yöntemler aynen C’deki işlevler gibi yazılır. Kısaca hatırlayalım:

Yöntemlerin belirli bir biçime uygun yazılmaları gerekir. Bu biçim şu şekildedir:

tür yöntemAdi(parametreListesi) {

}

`tür`: Yöntemin döndüreceği değerin türüdür. Örneğin verilen gerçel sayının karekökünü bulup döndüren bir yöntem yazacaksak, yöntemden geriye dönecek değerin türü gerçel sayı tipinde olacaktır. O zaman bu alana gerçel sayı türünü ifade eden double yazarız. Eğer yöntemin geriye herhangi bir değer döndürmesi beklenmiyorsa, bu alana değer dönmeyeceğini belirtmek üzere `void` yazılır.

`yöntemAdi`: Yönteme verilen addır. Yöntem adı, yöntemin ne iş yaptığını ifade edecek şekilde özenle seçilmelidir. Böylece hem kodun okunurluğu sağlanabilir hem de daha sonra hangi yöntemin kullanılacağına kolaylıkla karar verilebilir. Örneğin karekök bulan yöntem için `karekokBul` adı kullanılabilir.

`parametreListesi`: Bir yöntem, kendisine verilen değerler üzerinde çalışabilir. Karekök bulan yöntemin karekökünü bulacağı değer o yöntemin parametresidir. Parametre listesi, sıfır ya da daha çok parametreden oluşur. Her parametrenin önce türü sonra adı yazılır ve parametrelerin arasına “,” (virgül) koyulur.

Bu bilgiler ışığında, verilen iki tamsayının ortak bölenlerinin en büyüğünü bulan işlevi şu şekilde tanımlayabiliriz:

```
int obeb(int sayi1, int sayi1) {
      // buraya obeb bulma algoritması kodlanacak
}
```

Benzer şekilde, nesnenin durumunu ekrana yazdıran, geriye herhangi bir değer döndürmeyen yöntemi de şu şekilde tanımlayabiliriz:

```
void yaz() {
```

```
        // buraya nesnenin durumunu ekrana yazan kodlar yazılacak
}
```

Aşağıda **`Zaman`** sınıfına ait Java kodu görülmektedir:

```
class Zaman {

        int saat;
        int dakika;
        int saniye;

        /**
         * Her çağrıldığında nesneyi bir saniye sonrasına götüren yöntem.
         * saniye 59, dakika 59, saat ise 23'ten büyük olamaz.
         */
        void ilerle() {
                saniye++;
                if (saniye == 60) {
                        saniye = 0;
                        dakika++;
                        if (dakika == 60) {
                                dakika = 0;
                                saat++;
                                if (saat == 24) {
                                        saat = 0;
                                }
                        }
                }
        }
}
```

Kod 5-1. Zaman sınıfı

Görüldüğü gibi özellikleri sınıfın içinde birer değişken olarak, davranışı ise yine sınıfın içinde bir işlev olarak kodladık. **`Zaman`** sınıfının bir sınıf olduğunu belirten anahtar sözcük ise **`class`** sözcüğü. Bu kodu **`Zaman.java`** adlı bir dosyaya kaydettikten sonra derleyebiliriz.

Şu anda sözdizim kuralları açısından doğru ve geçerli bir sınıf yazmış durumdayız ancak henüz bu sınıf mantıksal açıdan kullanılabilir değil. Çünkü bu sınıfa ait bir nesne oluşturduğumuzda nesnemizin durumunu birer saniye ilerleterek değiştirebileceğiz fakat herhangi bir anda nesnenin durumunu öğrenebileceğimiz bir davranış tanımlamış değiliz. Başka bir deyişle zamanı ilerletebilir ama değerini öğrenemeyiz; "saat kaç?" diye soramayız. **Zaman** sınıfından bir nesneye bunu sorabilmemiz için, önce sınıfın içine bu soruyu yanıtlayabilecek davranışı tanımlamamız gerekiyor:

```
/**
 * Çağrıldığı zaman nesnenin o anki durumunu ekrana yazar.
 */
void zamaniYaz() {
      System.out.println("Zaman: " + saat + ":" + dakika + ":" + saniye);
}
```

Kod 5-2. Zaman nesnesinin durumunu soran yöntem

zamaniYaz() yöntemini, **Zaman** sınıfına ait herhangi bir nesnenin durumunu ekrana yazdıran bir yöntem olarak kodladık. Şimdi **Zaman** sınıfına ait bir nesne oluşturup bu nesneyi kullanan çalışabilir bir program elde edebilmek için **main** yöntemini yazalım.

main yöntemini **Zaman** sınıfının içine yazabiliriz. Ancak **Zaman** sınıfı kendi başına düşünüldüğünde, çalıştırılabilir bir sınıf olmak zorunda değildir. Bu sınıf, yalnızca zaman bilgisini ifade edebilen nesneler oluşturulabilmesi için yazılmış bir tanımdır. Başka bir deyişle, **Zaman** sınıfına ait kodların içinde bulmayı beklediğimiz yöntemler yalnızca bu sınıfa ait davranışlar olmalıdır. Öyleyse sırf örneğimizi çalıştırabilmek için **Zaman** sınıfının içine, aslında kendi davranışları ile ilgisi olmayan **main** yöntemini yazmak güzel bir yaklaşım olmayacaktır. **main** yöntemi çalıştırılabilir bir program elde etmek üzere yazılan bir yöntem olduğundan ve Java'da bütün kodların mutlaka bir sınıfın içine yazılması zorunlu olduğundan, bu yöntemi **Program** adlı bir sınıfın içine yazmak daha iyidir.

Örneklerde `main` yöntemini her zaman `Program` adlı bir sınıfın içine, diğer bütün sınıfları ise bütünlüklerini koruyacak, yalnızca kendi özellik ve davranışlarını içerecek şekilde kodlayacağım.

Aşağıda **`Program`** sınıfının kodu görünüyor:

```
class Program {
      public static void main(String[] args) {
            Zaman zamanNesnesi = new Zaman();
            zamanNesnesi.ilerle();
            zamanNesnesi.zamaniYaz();
      }
}
```

Kod 5-3. Zaman sınıfını kullanan Program sınıfı

Program adlı sınıfı çalıştırdığımızda ekranda aşağıdaki çıktı oluşacaktır:

```
Şu anki zaman: 0:0:1
```

Şimdi bu çıktının nasıl oluştuğuna bakalım:

Daha önce de belirttiğimiz gibi **`main`** yöntemi programın başlangıç noktasıdır. JVM, adını verdiğimiz sınıfın içinde **`main`** yöntemini bulur ve **`main`** yönteminin ilk satırından başlayarak programı işletir. **`main`** yönteminin ilk satırında, **`zamanNesnesi`** adı ile **`Zaman`** sınıfından bir nesne oluşturulmaktadır (nesne oluşturma ile ilgili ayrıntılı bilgi daha sonra verilecektir). Daha sonra bu nesneye **`ilerle()`** iletisi gönderilerek **`Zaman`** sınıfına tanımlanmış olan **`ilerle()`** davranışının gerçekleştirilmesi istenmektedir. **`ilerle()`** davranışının gerçekleştirilmesi **`ilerle()`** yönteminin işletilmesiyle olur ve bu yöntem nesnenin **`saniye`** niteliğinin değerini 1 arttırmaktadır. Nesne ilk oluşturulduğunda, niteliklerinin değerlerinin herbirinin değeri 0 (sıfır)'dır (**`Zaman`** sınıfının bütün nitelikleri **`int`** temel türündedir ve **`int`** temel türünün varsayılan değeri 0 (sıfır)'dır). **`ilerle()`** yöntemi bir kez çalıştığında, **`saniye`** niteliğinin değeri 1 olur. **`main`** yönteminin bir sonraki satırında ise **`zamanNesnesi`**'nin **`zamaniYaz()`** yöntemi çağrılmaktadır ve bu yöntem, niteliklerin o anki değerlerinden oluşturduğu dizgiyi ekrana yazmaktadır.

5.2 Nesne Oluşturma

Sınıfın bir tanım, nesnenin ise o tanıma göre oluşturulmuş fiziksel bir varlık olduğunu tekrar hatırlayalım. Bu ifade üzerinde biraz durursak şu sonuca varabiliriz: Nesne oluşturulmadığı sürece bellekte fiziksel olarak oluşturulmuş bir varlık yoktur; sınıfın nitelikleri için bellekten yer ayrılmamıştır. Örneğin, **`Zaman`** sınıfına ait bir nesne oluşturulmadığı sürece **`saat`**, **`dakika`** ve **`saniye`** nitelikleri yalnızca kağıt üzerinde birer tanımdır. Nesne oluşturulduğu andan itibaren ise bu niteliklerin her birisi için bellekten 32'şer ikil uzunluğunda (**`int`** türü 32 ikil uzunlukta), toplam 32 x 3 = 96 ikil uzunlukta yer ayrılacaktır.

5.2.1 `new` işleci

Herhangi bir sınıfa ait bir nesne oluşturulabilmesi için **`new`** işleci kullanılır. **`new`**, belirtilen sınıfa ait bir nesne oluşturup bu nesnenin bellekteki adresini (referansını) döndürür.

Kod 5-3'e tekrar bakarsak, **`main`** yönteminin ilk satırında

```
Zaman zamanNesnesi = new Zaman();
```

Kod 5-4. Zaman sınıfına ait zamanNesnesi adlı nesnenin oluşturulması - 1

yazmaktadır. Bu satırı iki farklı satır halinde tekrar yazalım:

```
Zaman zamanNesnesi;
zamanNesnesi = new Zaman();
```

Kod 5-5. Zaman sınıfına ait zamanNesnesi adlı nesnenin oluşturulması - 2

Şimdi de Java'da değişken tanımlama ve ilk değer atama işlemlerinin nasıl yapıldığını hatırlayalım. Örneğin bir tamsayı değişken tanımlamak ve o değişkene 5 değerini atamak için;

```
int sayi;
sayi = 5;
```

yazabileceğimiz gibi, tanımı yaptığımız satırda atama işlemini yaparak kısaca

```
int sayi = 5;
```

de yazabiliriz.

Bu bilgiyi hatırladıktan sonra Kod 5-5'e geri dönelim:

İlk satırda, **`Zaman`** sınıfına ait bir değişken tanımı yapılıyor; değişkenin türü **`Zaman`**, adı ise **`zamanNesnesi`** olarak seçilmiş. İkinci satırda ise **`zamanNesnesi`** adlı değişkene bir değer atanıyor. Atanan değer, **`new Zaman()`** deyiminden dönen değer (`zamanNesnesi = new Zaman()`). Bu sonuç ise **`Zaman`** sınıfına ait, **`new`** işleci ile bellekte oluşturulan nesnenin adresi. Yani bu satır işletildikten sonra **`zamanNesnesi`** adlı değişkenin içinde, bellekteki bir nesnenin (bu nesne **`Zaman`** sınıfına ait) adresi bulunuyor.

Kod 5-4'te ise, aynı işlem tek satırda yazılmış. Aynı satırda hem **`Zaman`** sınıfına ait nesnenin adresini tutabilecek bir değişken tanımı yapılmış, hem de **`Zaman`** sınıfından bir nesne oluşturulup, nesnenin adresi tanımlanan bu değişkenin içine atanmış.

5.2.2 Referans Tür

`new` işleci ile bir nesne oluşturulduğunu ve bu nesnenin bellekteki adresinin döndürüldüğünü söyledik. Döndürülen adres, oluşturulan nesneye daha sonra erişmek amacıyla referans olarak kullanılacaktır. Bu referansı saklayabilmek için de bir değişkene gereksinim vardır. İçerisinde referans (nesne adresi) saklayabilen türden değişkenlere referans türden değişken ya da Nesne Tutamaçı (*Object Handle*) adı verilir ve varsayılan değer olarak **`null`** değerini alırlar. **`null`**, bir nesne tutamaçının içinde o anda herhangi bir nesnenin adresi olmadığını belirten değerdir.

5.2.3 Yığın (*Heap*), Yığıt (*Stack*) ve Çöp Toplayıcı (*Garbage Collector*)

JVM'den bahsederken sanal donanımın 4 kesimden oluştuğunu ve bu kesimlerden birinin yığın (*heap*), birinin de yığıt (*stack*) olduğunu söylemiştik. Şimdi yığın ve yığıtın ne olduğunu biraz daha açalım:

JVM, oluşturulan değişken ve nesneleri yönetmek üzere bilgisayarın belleğini kullanır (32-ikillik JVM'nin en fazla 4GB bellek kullanabileceğini belirtmiştik). Ancak değişkenleri ve nesneleri yönetirken farklı bellek alanları ve algoritmalardan faydalanır.

Bir yöntem çağrıldığı zaman, o yönteme ait yerel değişkenler ile yöntemin parametreleri, yöntemin dönüş adresi ve dönüş değeri ile birlikte yığıt adı verilen bellek alanında oluşturulur. Yöntemin çalışması sona erdiği zaman bu bilgiler yığıttan atılır (bu bilgilere tekrar erişilemez). Değişkenlerin tanım alanları ve yaşam süreleri yöntem ile sınırlıdır. Yöntem yeniden çağrıldığında bütün bu alanlar yığıtta yeniden ve yeni değerleri ile oluşturulacaktır.

Nesneler ise yığında oluşturulurlar; **`new`** ile yeni bir nesne oluşturulduğu zaman, JVM bu nesneyi yığın adı verilen bellek alanında saklar. Nesnenin adresi ise yığıtta bulunan referans türdeki bir değişkenin içinde saklanır. Bu durumda, nesneyi oluşturan kod satırının içinde bulunduğu yöntem sona erdiğinde nesnenin referansını saklayan değişken yığıttan atılır. Ancak nesnenin kendisi hala bellektedir. Bir Java programının çalıştığı süre boyunca oluşturulan bütün nesneler yığında saklanır.

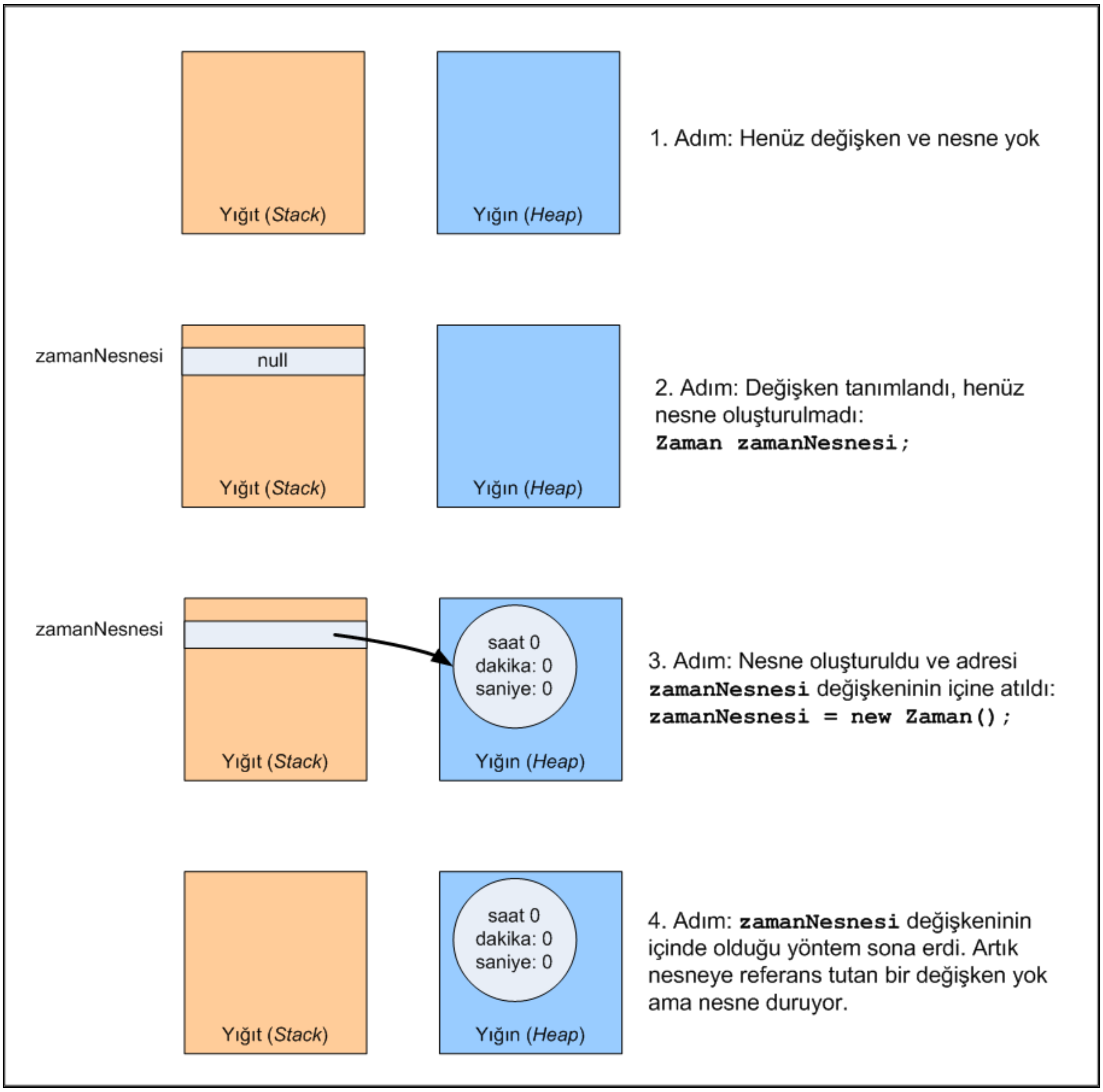
Yığının yönetilmesi için JVM'ler farklı algortimalar kullanırlar. JVM yığının yeterince dolduğuna karar verdiği zaman, artık kullanılmayan nesnelerin (bir nesnenin kullanılmıyor olması, o nesnenin referansını saklayan hiçbir değişken olmaması anlamına gelir) kullandıkları bellek alanlarını tekrar kazanabilmek için Çöp Toplayıcı adlı görevi devreye sokar. Çöp Toplayıcı, yığında bulunan ve kendisine hiç referans olmayan nesneleri (ki bunlara çöp denilebilir) bularak bunları ortadan kaldırır ve bunların kullanmakta oldukları bellek alanlarını yeniden yığına kazandırır. Çöp Toplayıcının ne zaman çalışacağına JVM karar verir. Java programlarının çalışması sırasında Çöp Toplayıcı hiç devreye girmeyebilir de.

Bellek alanlarının bu şekilde JVM tarafından yönetiliyor olması, C ya da C++ gibi dillerle programlama yaparken programcıyı yoran bellek yönetimi kodlarını ortadan kaldırmaktadır. Ancak yine de şunu söylemekte fayda

vardır: nesne oluşturmak ve kullanılmayan nesneleri bularak bunların tükettiği bellek alanlarını yeniden kazanmak işleyici ve bellek açısından maliyetli bir işlemdir programcının gereksiz nesne oluşturmaktan kaçınması gerekir.

Bellek yönetiminin JVM tarafından gerçekleştiriliyor olması programcının işini kolaylaştırıyor olabilir ancak programcı kendi programının başarımını düşürmemek için dikkatli kod yazmalıdır.

Aşağıdaki şekilde, nesne oluşturulması aşamalarında belleğin durumu görülmektedir.



Şekil 5-1. Nesne oluşturulması sırasında belleğin görünümü

Şekilde, herhangi bir yöntem çağrılmadan önce belleğin durumu 1. adımda gösterilmiştir. 2. adımda referans değişken tanımlanmaktadır. Şekilde de görüldüğü gibi, **`new`** işleci işletilmediği sürece bellekte nesne yoktur ve referans değişkenin ilk değeri "**`null`**"dur. **`new`** ile nesne oluşturulup nesnenin adresi referans değişkene aktarıldıktan sonra, yığındaki nesneye referansların sayısı (nesneyi işaret eden okların sayısı) 1 olur. Yöntem sona erip yığıttaki değişken ortadan kalktıktan sonra ise nesneye referansların sayısı 0 (sıfır)'dır ve bu nesne artık Çöp Toplayıcı tarafından yok edilmeye adaydır.

5.2.4 Kurucular (*Constructors*)

Nesnenin ancak **`new`** işleci işletildikten sonra bellekte var olduğunu gördük. Koda dikkatli bakarsak, **`new`** işlecinden hemen sonra gelen ifadenin **`Zaman()`** olduğunu, **`Zaman`**'ın ise nesnesini oluşturduğumuz sınıfın adı olduğunu görürüz. Ayrıca **`Zaman`**'ın hemen yanındaki **`()`** işaretleri, Java dilinde yöntem çağrılarını akla getirmektedir. Kod 5-3'e tekrar bakarsak, **`zamanNesnesi`** değişkenini kullanarak **`ilerle`** yöntemini çağıran deyimin

```
zamanNesnesi.ilerle()
```

şeklinde kodlandığını görürüz. **`()`** işaretlerinin yöntem çağrılarında kullanıldığını düşündüğümüzde, **`new Zaman()`** ifadesini **`Zaman`** adlı yöntemi çağıran bir ifade olarak görebiliriz. Başka bir deyişle **`new`** işleci, sınıf adı ile aynı isme sahip özel bir yöntemi çağırmaktadır gibi bir yorum yapabiliriz.

İşte bu özel işleve kurucu (*constructor*) adı verilir. Kurucu, bir nesne oluşturulurken mutlaka ilk olarak ve yalnızca bir defa çalışan, çalışması tamamlandığında ise yığında artık bir nesnenin var olduğu özel bir yöntem olarak düşünülebilir. Bir kurucuyu sınıfın içindeki diğer yöntemlerden ayıran iki temel özelliği vardır:

1. Sınıf ile aynı addadır.

2. Döndürebileceği değer alanı tanımlı değildir. Burada tanımlı olmamaktan kasıt, kurucunun değer döndürmeyen (**`void`**) bir yöntem olması değil, böyle bir tanımın var olmamasıdır.

Örneğin, **`Zaman`** sınıfı için tanımlanacak bir kurucu şu şekilde kodlanabilir:

```
Zaman() {
      // buraya kurucunun yapması istenen işlemler kodlanmalı
}
```

Kod 5-6. Zaman sınıfının kurucusu

Kod 5-6'da görüldüğü gibi, kurucunun adı sınıf adı ile aynı ve kurucunun döndüreceği değerin türünün yazılması gereken alanda hiçbirşey yazmıyor.

Şimdi hemen akla gelen iki soruyu yanıtlamaya çalışalım:

Kurucu niye yazılır?

Nesneye yönelik programlamada nesnelerin durumlarının kullanıldığını hatırlayalım. Öyleyse programın herhangi bir anında, bir nesnenin durumunun anlamlı ve tutarlı olması gerekir, aksi taktirde programda yanlış ya da anlamsız durumlar oluşabilir. Örneğin, **`Zaman`** programımızda **`saat`** niteliğinin değerinin hiçbir zaman 0 – 23 aralığının dışında, **`dakika`** ve **`saniye`** niteliklerinin de hiçbir zaman 0 - 59 aralığının dışında olmaması gerekir. **`Zaman`** sınıfının davranışlarını bunu sağlayacak şekilde kodlarız ve nesnenin durumunun tutarsız hale gelmesine izin vermeyiz. Ancak nesne ilk oluşturulduğu anda da durumunun anlamlı olması gerekir. Eğer **`Zaman`** örneğimizde 0 (sıfır) değeri **`saat`**, **`dakika`** ya da **`saniye`** niteliklerinden herhangi birisi için anlamsız olsaydı, nesne oluşturulduktan hemen sonra nesnenin durumu anlamsız olacaktı. Çünkü niteliklerin hepsi **`int`** türünde tanımlanmış ve **`int`** türünün varsayılan değeri 0 (sıfır).

Kurucunun rolü, nesnenin oluşturulması sırasında ilk anlamlı durumuna atanmasını sağlamaktır. İlk olarak kurucunun çalışacağı kesin olduğundan ve kurucu sonlandığında nesne oluşturulmuş olacağından, nesnenin ilk durumunun anlamlı olmasını sağlayacak kodun yazılabileceği yer kurucudur.

Kurucu yazmazsak ne olur?

Kod 5-1'de Zaman sınıfının içine kurucu yazmadığımız halde, Kod 5-3'te

```
Zaman zamanNesnesi = new Zaman();
```

şeklinde kodlanmış olan program için derleyici hata vermedi ve program sorunsuz çalıştı. Bu nasıl olabilir?

Eğer bir sınıf için kurucu yazılmazsa, kaynak kod derlenmeden hemen önce derleyici tarafından varsayılan kurucu (*default constructor*) üretilir ve sınıf öyle derlenir. Varsayılan kurucu, hiçbir parametre almayan ve **{ }** işaretlerinin arası boş olan bir kurucu olarak düşünülebilir:

```
Zaman() {
}
```

Sınıfın içine bizim bir kurucu yazmamız durumunda ise derleyici artık herhangi bir şekilde kurucu üretmez.

Kurucu, nesnenin nasıl oluşturulacağını söyleme olanağı sağladığı kadar, nasıl oluşturulamayacağını söyleme olanağı da sağlar. Eğer sınıfın içine bir kurucu tanımlanmışsa, derleyicinin artık varsayılan kurucu üretmeyeceğini söyledik. Eğer tanımlanan kurucu bir takım parametreler alıyorsa bu, o sınıfa ait bir nesne oluşturulabilmesi için mutlaka bu parametreler ile kurucunun çağrılmasını zorlayacaktır. Örneğin **Zaman** sınıfı için aşağıdaki gibi bir kurucu tanımlarsak, artık

```
Zaman zamanNesnesi = new Zaman();
```

şeklinde kod yazamayız. Çünkü artık derleyici tarafından varsayılan kurucu üretilmeyecektir ve **Zaman** sınıfının içinde hiç parametre almayan bir kurucu tanımı da yoktur.

```
/**
 * Nesnenin ilk durumunu verilen değerlere çeker
 *
 * @param st      Saat için bağlangıç değeri
 * @param dk      Dakika için başlangıç değeri
 * @param sn      Saniye için başlangıç değeri
 */
```

```
Zaman(int st, int dk, int sn) {
      saat = st;
      dakika = dk;
      saniye = sn;
}
```

Kod 5-7. Zaman sınıfında parametreli kurucu

Bu durumda, eğer bir **`Zaman`** nesnesi oluşturulmak isteniyorsa, şuna benzer şekilde kod yazılması gerekir:

```
Zaman zamanNesnesi = new Zaman(0, 0, 0);
```

5.2.5 `this` anahtar sözcüğü

`Zaman` sınıfına ait parametreli kurucuyu yazarken niteliklere doğrudan erişebildik. Çünkü bu niteliklerin tanımlı oldukları alan sınıf tanımını başlatan **`{`** işareti ile sınıf tanımını bitiren **`}`** işareti arasıdır.

Aşağıda parametreli kurucunun kodu görülmektedir. Dikkat edilirse, parametre adları **`st`**, **`dk`** ve **`sn`** olarak seçilmiş, böylece **`saat`**, **`dakika`** ve **`saniye`** adlı nitelikler ile parametrelerin aynı isimde olmaları engellenmiştir. Eğer parametre adları da **`saat`**, **`dakika`** ve **`saniye`** olarak seçilirse ne olur? Bu durumda kurucu içinde örneğin **`saat`** adı ile ancak saat adlı parametrenin kendisine erişilebilir; **`saat`** niteliğine bu şekilde erişilemez.

```
Zaman(int st, int dk, int sn) {
      saat = st;
      dakika = dk;
      saniye = sn;
}
```

Kod 5-8. Parametreli kurucu

Nitelik ile parametrenin adları aynı ise bu ikisi arasında ayrım yapabilmek için **`this`** anahtar sözcüğü kullanılır:

```
Zaman(int saat, int dakika, int saniye) {
      this.saat = saat;
      this.dakika = dakika;
      this.saniye = saniye;
```

```
}
```

Kod 5-9. this anahtar sözcüğünün kullanılışı

Bu kodu açıklayalım: parametre adları **`saat`**, **`dakika`** ve **`saniye`** olduğu için, kurucunun içinde örneğin **`saat`** yazıldığında **`saat`** adlı parametrenin değerine ulaşılmaktadır. Gelen parametrenin değerinin nitelik olan **`saat`**'in içine atanabilmesi için nitelik olan **`saat`**, **`this`** anahtar sözcüğü ile belirtilmiştir:

```
...
this.saat = saat;
...
```

`this` anahtar sözcüğü, deyimin işletildiği anda hangi nesne ile çalışılıyorsa o nesnenin kendisine referanstır. Diyelim ki **`Zaman`** sınıfına ait **`zaman1`** ve **`zaman2`** nesneleri oluşturuluyor. **`this`** anahtar sözcüğü,

```
Zaman zaman1 = new Zaman(3, 16, 34);
```

kodu işletilirken **`zaman1`** referansı ile gösterilen nesnenin adresini,

```
Zaman zaman2 = new Zaman(15, 0, 0);
```

kodu işletilirken ise **`zaman2`** referansı ile gösterilen nesnenin adresini belirtir.

Bir sınıfın içine yazılan kodlarda, nesne referansları üzerinden erişilebilecek nitelik ya da yöntem çağrılarının kodlandıkları alanlarda aslında **`this`** anahtar sözcüğü gizli olarak (*implicitly*) zaten kullanılmaktadır. Eğer biz bu anahtar sözcüğü kodlarsak, yalnızca açıkça (*explicitly*) yazmış oluruz. Başka bir fark yoktur.

Parametre adı ile nitelik adının aynı olduğu durumda ise **`this`** anahtar sözcüğünün kullanılması zorunludur. Çünkü yöntemin içindeki kod kesiminde, parametre kendisi ile aynı addaki niteliğe erişimi engellemektedir.

5.2.6 Kurucu Aşırı Yükleme (*Constructor Overloading*)

Bir sınıf için birden fazla kurucu tanımlanabilir. Kurucuları birbirlerinden ayıracak olan, kurucuların parametreleridir. Parametre sayıları ya da

türlerinin farklı olması sayesinde hangi kurucunun çağrıldığı anlaşılabileceğinden bir belirsizlik oluşmaz.

Aşırı yükleme (*overloading*) ifadesi ile anlatılmak istenen, tek bir ismin birden fazla kurucu için kullanılması, kısaca ***ismin*** *aşırı yüklenmesi*dir.

Zaman sınıfına iki tane kurucu yazalım. Bunlardan ilki oluşturulacak bir **Zaman** nesnesinin bütün niteliklerine sıfır değerini, ikincisi ise bu niteliklerin değerlerine parametre olarak verilen değerleri atasın.

```
Zaman() {
      saat = 0;
      dakika = 0;
      saniye = 0;
}
Zaman(int st, int dk, int sn) {
      saat = st;
      dakika = dk;
      saniye = sn;
}
```

Kod 5-10. Zaman sınıfı için kurucular

Zaman sınıfının içine birden fazla kurucu tanımlamış olduk. Bu durum herhangi bir derleme hatasına neden olmayacaktır. Çünkü iki kurucu, parametre listelerinin farklı olması sayesinde rahatlıkla birbirlerinden ayrılabilmektedir.

Zaman sınıfımızın birden fazla kurucu içerecek şekilde kodlanmış son haline tekrar bakalım:

```
class Zaman {

      int saat;
      int dakika;
      int saniye;

      /**
       * Nesnenin ilk durumunu 0, 0, 0 yapar.
       */
      Zaman() {
```

```
		saat = 0;
		dakika = 0;
		saniye = 0;
	}

	/**
	 * Nesnenin ilk durumunu verilen değerlere çeker
	 *
	 * @param st	Saat için bağlangıç değeri
	 * @param dk	Dakika için başlangıç değeri
	 * @param sn	Saniye için başlangıç değeri
	 */
	Zaman(int st, int dk, int sn) {
		if (st >= 0 && st < 24) {
			saat = st;
		}
		if (dk >= 0 && dk < 60) {
			dakika = dk;
		}
		if (sn >= 0 && sn < 60) {
			saniye = sn;
		}
	}

	/**
	 * Her çağrıldığında nesneyi bir saniye sonrasına götürür.
	 * saniye 59, dakika 59, saat ise 23'ten büyük olamaz.
	 */
	void ilerle() {
		saniye++;
		if (saniye == 60) {
			saniye = 0;
			dakika++;
			if (dakika == 60) {
				dakika = 0;
				saat++;
				if (saat == 24) {
					saat = 0;
				}
			}
```

```
            }
      }

      /**
       * Çağrıldığı zaman nesnenin o anki durumunu ekrana yazar.
       */
      void zamaniYaz() {
            System.out.println("Şu anki zaman: " +
                        saat + ":" + dakika + ":" +   saniye);
      }
}
```

Kod 5-11. Zaman sınıfı

5.2.7 Kurucuların Birbirini Çağırması

Bir sınıfın içine birden fazla kurucu yazdığımızda, bazen aynı işi tekrar tekrar kodluyor olabiliriz. Örneğin **Zaman** sınıfının içine 3 farklı kurucu yazalım. Birisi yalnızca **saat** niteliğinin değerini parametre olarak alsın, birisi hem **saat** hem **dakika** için parametre alsın, sonuncusu da bütün nitelikler için parametre alsın:

```
class Zaman {

      int saat;
      int dakika;
      int saniye;

      Zaman(int saat) {
            if (saat >= 0 && saat < 24) {
                  this.saat = saat;
            }
      }
      Zaman(int saat, int dakika) {
            if (saat >= 0 && saat < 24) {
                  this.saat = saat;
            }
            if (dakika >= 0 && dakika < 60) {
                  this.dakika = dakika;
            }
      }
```

```
      Zaman(int saat, int dakika, int saniye) {
            if (saat >= 0 && saat < 24) {
                  this.saat = saat;
            }
            if (dakika >= 0 && dakika < 60) {
                  this.dakika = dakika;
            }
            if (saniye >= 0 && saniye < 60) {
                  this.saniye = saniye;
            }
      }
}
```

Kod 5-12. 3 kuruculu Zaman sınıfı

Kurucular incelenirse, bazı kodların tekrar tekrar yazıldığı görülür. Bir kod kesiminin tekrarlanması istenen bir durum değildir. Çünkü kodun tekrarlanması, hem tutarsızlıklara neden olabilir hem de bakımı yapılacak kod miktarını arttırır. Bunun engellenebilmesi için, yazılan kodu kopyalamak yerine, kodun yazıldığı kesimi işlev haline getirip çağrılması düşünülebilir.

Örneğimizde, iki parametreli kurucu tek parametreli kurucunun içeriğini aynen kopyaladıktan sonra **`dakika`** ile ilgili kısmı eklemiş, üç parametreli kurucu ise iki parametreli kurucunun içeriğini aynen kopyalayıp **`saniye`** ile ilgili kısmı eklemiştir. Öyleyse iki parametreli kurucuda bir parametreli kurucuyu, üç parametreli kurucuda da iki parametreli kurucuyu çağırabilirsek kod tekrarından kurtulabiliriz.

Bunu yapabilmek için **`this`** anahtar sözcüğü kullanılır. **`this`** anahtar sözcüğü, nesnenin kendisine referans olduğu gibi, uygun parametreli kurucunun da işletilmesini sağlayabilir:

```
class Zaman {

      int saat;
      int dakika;
      int saniye;

```

```
    Zaman(int saat) {
        if (saat >= 0 && saat < 24) {
            this.saat = saat;
        }
    }
    Zaman(int saat, int dakika) {
        this(saat);
        if (dakika >= 0 && dakika < 60) {
            this.dakika = dakika;
        }
    }
    Zaman(int saat, int dakika, int saniye) {
        this(saat, dakika);
        if (saniye >= 0 && saniye < 60) {
            this.saniye = saniye;
        }
    }
}
```

Kod 5-13. this anahtar sözcüğü ile kurucuların çağrılması

Burada dikkat edilmesi gereken nokta, **`this()`** ile kurucu çağıran kod kesiminin, içinde bulunduğu kurucunun ilk deyimi olması zorunluluğudur. Eğer **`this()`** ilk deyim olarak yazılmazsa, derleyici hata verecektir.

5.2.8 Yöntem Aşırı Yükleme (*Method Overloading*)

Kurucularda olduğu gibi yöntemler de aşırı yüklenebilir.Aynı adlı birden fazla yöntem, parametre listelerindeki farklılıklar sayesinde birbirlerinden ayrılabilir. Parametre sayısılarının ya da türlerinin farklı olması yeterlidir.

Ancak burada akla gelebilecek bir başka soru var: Birden fazla kurucu yazmak istediğimizde aşırı yükleme kaçınılmaz, çünkü kurucular sınıf ile aynı adı taşımak zorunda. Peki sıradan yöntemler için neden aynı adı kullanalım ki? Yöntemlere istediğimiz gibi isim verebileceğimize göre aşırı yüklemeye neden gerek olsun?

Bu soruyu yanıtlamak için önce programın kaynak kodlarının yalnızca o kodu okuyan insanlar için anlamlı olduğunu hatırlayalım. Derleyici ya da JVM açısından, bir yöntemin adının **`xyztysdf`** olması ile **`zamaniYaz`** olması

arasında herhangi bir fark yoktur. Ancak biz insanlar için yöntem adları, yöntemin ne iş yaptığını anlatmaları bakımından önemlidir. Bir yöntemi adlandırırken o yöntemin yaptığı işi özetleyecek bir ad seçer, böylece kodun okunurluğunu arttırmayı ve programlamanın da daha kolay olmasını sağlamaya çalışırız. Eğer aynı işi farklı şekillerde gerçekleştiren ya da farklı parametre listeleri ile çalışarak aynı işi gerçekleştiren yöntemler yazmak istiyorsak, bu yöntemlerin herbiri için ayrı adlar belirlemek yerine, yapılan işi özetleyen tek bir adı kullanmak hem daha kolay hem daha anlamlıdır. Yöntem aşırı yüklemenin getirisi programcıya bu kolaylığı sağlamasından ibarettir. Bununla birlikte, programcı istiyorsa hala her yöntemine farklı ad vermekte de özgürdür.

`Zaman` sınıfımızın **`ilerle`** yöntemini aşırı yükleyelim. Daha önce yazdığımız **`ilerle`** yöntemi, her çağrıldığında zamanı bir saniye ilerletiyordu. Şimdi bir de zamanı verilen saniye kadar ilerleten yöntem kodlayalım:

```
/**
 * Verilen saniye kadar zamanı ilerletir
 *
 * @param sn      Zamanın kaç saniye ilerletileceği
 */
void ilerle(int sn) {
      saniye += sn;
      if (saniye > 59) {
            dakika += saniye / 60;
            saniye %= 60;
            if (dakika > 59) {
            saat += dakika / 60;
                  dakika %= 60;
                  if (saat > 23) {
                        saat %= 24;
                  }
            }
      }
}
```

Kod 5-14. Zamanı verilen saniye kadar ilerleten yöntem

Ve programımızı da aşırı yüklenmiş kurucuları ve yöntemleri kullanacak şekilde değiştirelim.

```
class Program {

      public static void main(String[] args) {
            Zaman zaman1 = new Zaman();
            Zaman zaman2 = new Zaman(14, 12, 34);

            zaman1.ilerle(12136);
            zaman1.zamaniYaz();

            zaman2.ilerle();
            zaman2.zamaniYaz();
      }
}
```

Kod 5-15. Zaman sınıfının aşırı yüklenmiş kurucu ve yöntemlerini kullanan program

Programın çıktısı şu şekilde olacaktır:

```
Şu anki zaman: 3:22:16
Şu anki zaman: 14:12:35
```

Çıktı 5-1. Program sınıfının çıktısı

Görüldüğü gibi programda iki ayrı **`Zaman`** nesnesi oluşturulmaktadır. **`zaman1`** adlı değişken, ilk nesnenin referansını tutmakta ve bu nesne parametre almayan kurucu ile oluşturulmaktadır. **`zaman2`** ise 3 parametreli kurucu ile oluşturulan nesnenin referansını tutmaktadır. **`zaman1`**, 12136 saniye ilerletilip zamanı ekrana yazması istendiğinde **`3:22:16`** değerini yazmış, **`zaman2`** ise 1 saniye ilerletilip ekrana **`14:12:35`** değerini yazmıştır. Hangi kurucunun ya da hangi **`ilerle`** yönteminin çalışacağında parametreler belirleyici olmuştur.

5.2.9 Diziler

Diziler, aynı türden birçok değişkeni bir arada tanımlamayı ve yönetmeyi sağlayan programlama araçlarıdır. Tek bir tanımla birçok değişken tanımlanır ve bunların herbirine erişim için aynı isim bir indisle birlikte

kullanılır. Dizi elemanlarına aynı isim üzerinden indis kullanılarak erişilebilmesi, bu elemanların bellekte **ardışık** olarak tutulmasıyla sağlanmaktadır. Örneğin 100 elemanlı bir tamsayı (**int**) dizisi oluşturulduğunda, herbir tamsayı elemanın uzunluğunun 4 *byte* (32 ikil) olduğu düşünülürse, 4 x 100 = 400 *byte* uzunluğunda bir bellek bloğu ayrılmış demektir.

Dizilerle çalışırken, diziyi gösteren değişkeni, dizinin kendisini, dizinin elemanlarını ve varsa o elemanların gösterdikleri nesneleri birbirlerinden ayırabilmek önemlidir.

Bir diziyi gösteren değişkenin tanımı şu şekilde yapılır:

dizinin_elemanlarının_türü[] dizininAdi;

Örnek:

```
int[] tamsayiDizisi;
Zaman[] zamanDizisi;
```

Bu tanım yapıldığında, henüz ortada dizi yoktur, bellekte dizi için ayrılmış yer yoktur. Sadece, bellekte bir dizi oluşturulduğunda, o dizinin bulunduğu adresi (dizinin referansını) tutabilecek büyüklükte (4 byte – 32 ikil) bir değişken tanımı yapılmıştır. Bu tanımın, mantıksal olarak

```
Zaman zamanNesnesi;
```

tanımından bir farkı yoktur. Her iki tanımlama da referans türden bir değişken tanımıdır. Dizi referansı tanımlandığında henüz bellekte bir dizi oluşturulmamış, nesne referansı tanımlandığında da henüz bellekte bir nesne oluşturulmamıştır.

Dizi,

```
tamsayiDizisi = new int[100];
zamanDizisi = new Zaman[50];
```

şeklinde dizinin kaç elemanlı olduğu belirtilerek oluşturulur. Dikkat edilirse dizi oluşturulurken **new** işleci kullanılmıştır. Buradan, dizilerin de nesneler gibi ele alındıkları ve yığında oluşturuldukları sonucunu çıkartabiliriz.

Bir dizi oluşturulduğunda, dizinin tanımlı olduğu elemanın türünden, dizinin boyu kadar değişken tanımlanmış demektir. Yani yukarıdaki dizi tanımlarını düşünürsek;

```
new int[100];
```

yazmakla

```
int sayi1;
int sayi2;
int sayi3;
....
```

şeklinde 100 adet tamsayı değişken tanımlamak arasında mantıksal açıdan bir fark yoktur. Öyleyse, dizinin herbir elemanının tamsayı değişken olduğunu düşünebiliriz. Tek fark, bu değişkenlere ayrı birer isimle değil, dizi referansı ve bir indis değeri kullanılarak erişiliyor olmasıdır.

Dizi elemanlarına erişim şu şekilde gerçekleştirilir:

```
tamsayiDizisi[0] = 5;
tamsayiDizisi[99] = -12;
```

Java'da dizi indisleri 0 (sıfır)'dan başlar. Dizinin ilk elemanı 0 indisli elemandır. Dolayısıyla dizinin son elemanı da (diziboyu – 1) indisli eleman olacaktır (100 elemanlı dizinin son elemanı 99 indisli elemandır).

Bir dizinin uzunluğu, dizinin **`length`** niteliği kullanılarak öğrenilebilir:

```
int dizininUzunlugu = tamsayiDizisi.length;
```

Bu bilgiler ışığında, tamsayi dizisini tanımlayıp elemanlarına sıradan değer atayan bir örnek kod kesimini şu şekilde yazabiliriz:

```
int dizi[];
dizi = new int[100];
for (int i = 0; i < dizi.length; ++i) {
      dizi[i] = i + 1;
}
```

Kod 5-16. Dizi elemanlarına erişim örneği - 1

Eğer dizi referans değişkenini tanımlamak ve diziyi oluşturmak için yazılan kodları tek satırda birleştirirsek, şu kodu elde ederiz:

```
int dizi[] = new int[100];
```

```
for (int i = 0; i < dizi.length; ++i) {
      dizi[i] = i + 1;
}
```

Kod 5-17.Dizi elemanlarına erişim örneği – 2

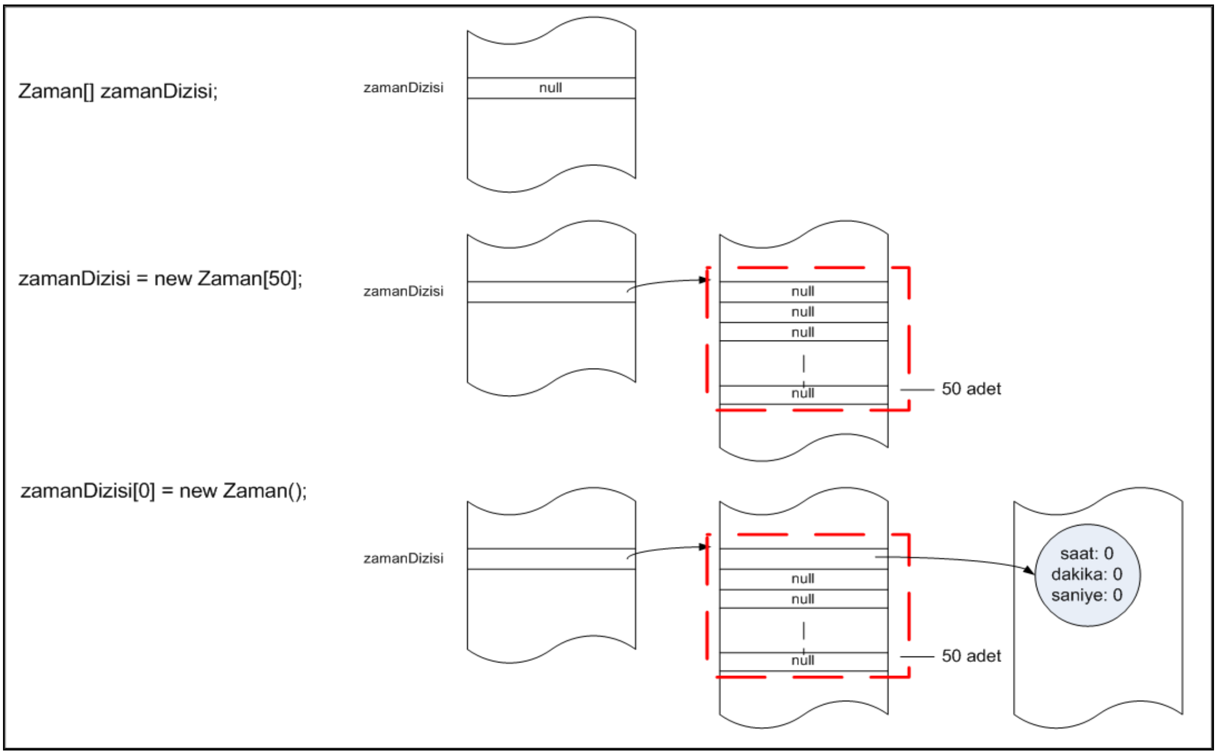
Şimdi bir soru soralım:

```
Zaman zamanDizisi = new Zaman[50];
```

tanımlaması yapıldığı zaman, bellekte kaç tane **`Zaman`** nesnesi oluşturulmuş olur? Yanıt 0 (sıfır). Çünkü bu tanımlama;

- Önce bellekte 4 byte uzunluğunda bir yer ayırıyor ve bu yere **`zamanDizisi`** adını veriyor (dizi referansı değişkeni).
- 50 adet, herbiri **`Zaman`** tipinde bir nesnenin adresini tutabilecek uzunlukta (4 byte) referans değişken için bellekten yer alıyor: 50 x 4 = 200 byte.
- Alınan bu yerin adresini **`zamanDizisi`** adlı değişkenin içine kopyalıyor.
- Dikkat edilirse henüz hiç **`Zaman`** nesnesi oluşturulmadı (kodun hiçbir yerinde **`new Zaman()`** gibi, kurucunun çağrılmasını sağlayacak bir kod yok). **`zamanDizisi`** adlı referansla ulaşılan bellek adresinde ise herbirinin değeri **`null`** olan 50 tane referans değişken arka arkaya duruyor.

Bunu aşağıdaki şekille açıklamaya çalışalım:



Şekil 5-2. Diziler ve bellekteki durum

Çok Boyutlu Diziler:

Diziler birden fazla boyutlu olabilir. Örneğin bir matrisi ifade etmek üzere iki boyutlu dizi tanımlanabilir.

```
int matris[][];
matris = new int[5][10];
```

Bu tanımlama, 5 satırı olan, her satırında 10 sütün bulunan bir matris oluşturmaktadır ve şu şekilde de yazılabilir:

```
int matris[][] = new int[5][10];
```

Matrisin elemanlarına erişim için iki indis kullanılır (idisler her boyut için 0 (sıfır)'dan başlar). Aşağıdaki kod kesimi matrisin ilk satırının ilk sütununa 3, son satırının son sütununa ise 5 değerini aktarmaktadır.

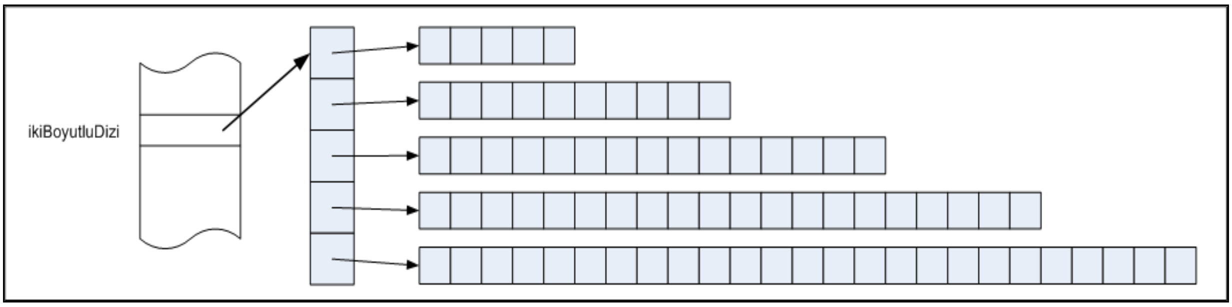
```
matris[0][0] = 3;
matris[4][9] = 5;
```

Çok boyutlu dizilerin her boyutu, ayrı ayrı da oluşturulabilir. Bu durumda, her satırında farklı sayıda sütun olan iki boyutlu bir dizi şu şekilde oluşturulabilir:

```
int ikiBoyutluDizi[][] = new int[5][];
```

```
ikiBoyutluDizi[0] = new int[5];
ikiBoyutluDizi[1] = new int[10];
ikiBoyutluDizi[2] = new int[15];
ikiBoyutluDizi[3] = new int[20];
ikiBoyutluDizi[4] = new int[25];
```

Yukarıdaki kod kesiminde, iki boyutlu bir dizi oluşturulmaktadır. İlk boyutu 5 elemanlıdır. İkinci boyutundaki eleman sayıları ise farklıdır. Sonuçta oluşan veri yapısı şekildeki gibidir:



Şekil 5-3. İki boyutlu dizi

5.3 Sarmalama (*Encapsulation*) İlkesi ve Erişim Düzenleyiciler (*Access Modifiers*)

Nesneye yönelik programlama yaklaşımı ile uygulama geliştirilirken bir takım ilkelere göre davranılır. Bu ilkeler kodun değiştirilebilirlik, yeniden kullanılabilirlik ve genişletilebilirlik gibi kalite kriterlerinin sağlanmasında çok önemli role sahiptirler.

NYP'nin 3 temel ilkesi/aracı vardır. Bunlardan ilki kodun değiştirilebilirliğini sağlamakta büyük katkısı olan sarmalama (*encapsulation*) ilkesidir.

5.3.1 Sarmalama İlkesi

Bir sınıfın içindeki niteliklerin, programın çalışması sırasında nesnelerin durumlarını oluşturduğunu gördük. Bir nesnenin durumunun ise her zaman anlamlı ve tutarlı olmak zorunda olduğunu söyledik. Nesne ilk oluşturulduğu anda nesnenin ilk durumunu anlamlı kılabilmek için de kuruculardan faydalandık.

`Zaman` örneğimizde, herhangi bir **`Zaman`** nesnesinin durumunun anlamsız (**`saat`** niteliğinin 0'dan küçük ya da 23'ten büyük, **`dakika`** ve **`saniye`** niteliklerinin ise 0'dan küçük ya da 59'dan büyük) olamayacağını biliyoruz. Çünkü bu niteliklerin değerleri kurucu ile 0 ya da başka anlamlı değerlere atandıktan sonra, **`ilerle`** yöntemleri ile denetimli bir şekilde değiştiriliyorlar. Bu yöntemler kullanıldığı sürece nitelikler hiçbir zaman belirlenen aralıkların dışında değerlere sahip olamazlar.

Ancak eğer **`Zaman`** sınıfının dışındaki herhangi bir kod kesiminde bu niteliklere erişim olanaklı ise bu söylediklerimiz geçerliliğini kaybedebilir.

> ***"Zaman sınıfının dışındaki herhangi bir kod kesimi", Zaman sınıfını başlatan ve bitiren { ve } işaretleri arasına <u>yazılmamış</u> olan kod kesimi anlamına gelmektedir.***
>
> ***"niteliklere erişim" ise kod kesiminde niteliğin adını yazarak kodlama yapmak anlamına gelmektedir. Örneğin Kod 5-18'de zaman adlı nesnenin saat niteliğine zaman.saat yazılarak "erişilmekte"dir.***

Örneğin **`Program`** sınıfının kodunu şu şekilde değiştirelim:

```
class Program {
      public static void main(String[] args) {
            Zaman zaman = new Zaman();
            zaman.saat = 25;
            zaman.dakika = 62;
            zaman.saniye = -5;
            zaman.zamaniYaz();
      }
}
```

Kod 5-18. Zaman sınıfı dışından niteliklere erişim

Program sınıfı çalıştırıldığında aşağıdaki çıktı oluşur:

```
Şu anki zaman: 25:62:-5
```

Görüldüğü gibi eğer **`Zaman`** sınıfından bir nesnenin niteliklerine, sınıfın içine tanımlanmış olan yöntemler dışında erişilip değer atanabilirse bu niteliklerin değerleri üzerindeki denetim ortadan kalkmış olur. Başka bir

deyişle, eğer bir sınıfa ait nesnelerin niteliklerinin anlamlı ve tutarlı değerlerde kalmalarını istiyorsak, bu niteliklerin değerlerini değiştirebilecek kod kesimlerinin mutlaka denetimli olmasını sağlamak zorundayız. Bu ise ancak niteliklerin değerlerini değiştirmek isteyen kod kesimlerinin, nesnenin ait olduğu sınıfın içine kodlanmış yöntemleri kullanmalarını zorlayarak yapılabilir. Örneğin, **`Zaman`** sınıfından bir nesnenin niteliklerinin değerlerini değiştirebilecek bütün yöntemlerin **`Zaman`** sınıfı içine kodlanması, bu sınıfın dışında kalan kod kesimlerinde ise (örneğin **`Program`** sınıfı) ancak ve ancak bu yöntemlerin kullanılmasının zorlanması gerekmektedir.

Böylece **`Zaman`** sınıfının nitelikleri yalnızca kurucular ve `ilerle` yöntemleri ile değiştirilebilir ve bu yöntemlerin içinde de niteliklerin değerlerinin anlamlı olacakları garanti edilmektedir.

Sarmalama ilkesi, bir sınıfa ait niteliklerin değerlerinin ancak o sınıfın içine kodlanmış yöntemler tarafından değiştirilebilmesi ve okunabilmesi ilkesidir. Böylece nesnelerin durumları her zaman anlamlı kalabilir.

Ayrıca niteliklere sınıf dışından erişimin olmaması, bir sınıf içindeki niteliklerin nasıl ve kaç adet olacağının da başka kod kesimlerinden saklanmış olması anlamına gelir. Bu durumda sınıfın içindeki kodlar başka sınıflar etkilenmeden değiştirilebilir. Bunu bir örnekle açıklayalım:

`Zaman` sınıfındaki niteliklerin diğer kod kesimlerinden erişilebilmesini kesin olarak engelleyebilirsek, **`Zaman`** sınıfının içinde hangi niteliklerin bulunduğunun uygulamanın geri kalan kısmı için bir önemi kalmaz. Örneğin, **`Zaman`** sınıfı içinde **`saat`**, **`dakika`** ve **`saniye`** olarak 3 adet nitelik tutmak yerine, yalnızca **`saniye`** niteliğini tutup gerekli olduğunda hesaplama yaparak **`dakika`** ve **`saat`** değerlerine ulaşabiliriz. **`Zaman`** sınıfına ait nesneler kullanan **`Program`** sınıfı, **`Zaman`** sınıfının kurucu, **`ilerle`** ve **`zamaniYaz`** yöntemlerini kullanmakta olduğundan, gene aynı şekilde çalışacaktır. **`Zaman`** sınıfının içinde yapılan değişiklikler hiçbir şekilde

`Program` sınıfını etkilememiş olacaktır. Aşağıda bu durumu örnekleyen kodlar görülmektedir.

```
class Zaman {

	int saniye;

	/**
	 * Nesnenin ilk durumunu 0, 0, 0 yapar.
	 */
	Zaman() {
		saniye = 0;
	}

	/**
	 * Nesnenin ilk durumunu verilen değerlere çeker
	 *
	 * @param st	Saat için bağlangıç değeri
	 * @param dk	Dakika için başlangıç değeri
	 * @param sn	Saniye için başlangıç değeri
	 */
	Zaman(int st, int dk, int sn) {
		if (st >= 0 && st < 24) {
			saniye += st * 60 * 60;
		}
		if (dk >= 0 && dk < 60) {
			saniye += dk * 60;
		}
		if (sn >= 0 && sn < 60) {
			saniye += sn;
		}
	}

	/**
	 * Her çağrıldığında nesneyi bir saniye sonrasına götürür.
	 * saniye 59, dakika 59, saat ise 23'ten büyük olamaz.
	 */
	void ilerle() {
		saniye++;
	}
```

```
	/**
	 * Verilen saniye kadar zamanı ilerletir
	 *
	 * @param sn      Zamanın kaç saniye ilerletileceği
	 */
	void ilerle(int sn) {
		saniye += sn;
		if (saniye < 0) {
			saniye = 0;
		}
	}
	/**
	 * Çağrıldığı zaman nesnenin o anki durumunu ekrana yazar.
	 */
	void zamaniYaz() {
		int saat = saniye / (60 * 60);
		int dakika = (saniye / 60) % 60;

		System.out.println("Şu anki zaman: " +
				saat + ":" + dakika + ":" +   (saniye % 60));
	}
}
```

Kod 5-19. Değiştirilmiş Zaman sınıfı

```
class Program {
	public static void main(String[] args) {
		Zaman zaman1 = new Zaman();
		Zaman zaman2 = new Zaman(14, 12, 34);

		zaman1.ilerle(12136);
		zaman1.zamaniYaz();

		zaman2.ilerle();
		zaman2.zamaniYaz();
	}
}
```

Kod 5-20. Değiştirilmiş Zaman sınıfını kullanan Program sınıfı

```
Şu anki zaman: 3:22:16
Şu anki zaman: 14:12:35
```

Çıktı 5-2. Program sınıfının çıktısı

Kod 5-19'da, **`Zaman`** sınıfındaki **`dakika`** ve **`saat`** niteliklerinin silinerek yalnızca **`saniye`** niteliğinin kaldığı görülmektedir. **`Zaman`** sınıfının yöntemleri de yalnızca **`saniye`** niteliği ile çalışacak şekilde değiştirilmiş durumdadır. Kod 5-20'deki **`Program`** sınıfında ise hiçbir değişiklik yapılmamıştır (Kod 5-15 ile aynıdır) ve Çıktı 5-2'de görülen çıktı, Çıktı 5-1'deki ile aynıdır.

Sarmalama ilkesi kodun değiştirilebilirliğini sağlar ve bakımını kolaylaştırır.

5.3.2 Paket Kavramı

Erişim düzenleyicilerden hemen önce paket kavramını anlatmamız gerekiyor. Böylece erişim düzenleyicilerden bahsederken, paket erişimini belirleyen düzenleyiciyi de açıklayabileceğiz.

Önce bir sınıfı nasıl adlandırdığımızı düşünelim: genellikle, sınıfın ilişkili olduğu kavramın adını kullanıyor, ilgisiz adlandırmalardan kaçınarak kodun okunurluğunu sağlamaya özen gösteriyoruz. Örneğin zamanı ifade etmek üzere tasarladığımız sınıfın adını **`Zaman`** seçtik. Bir öğrenci yönetim sistemi geliştiriyor olsaydık, büyük olasılıkla sınıf adlarımız **`Ogrenci`**, **`Ders`**, **`Ogretmen`**.. gibi adlar olacaktı.

Java ile programlama yapan herkesin aynı yaklaşımı benimsediğini düşünürsek, benzer kavramları karşılamak üzere benzer isimler kullanılacağını görürürüz. Bu durumda, dünyanın heryerinde aynı sınıf isimleri kullanılıyor olabilir.

Geliştireceğimiz uygulamalar kaçınılmaz olarak Java kütüphanelerinden faydalanacaktır. Bu kütüphaneler ise başka başka insanlar tarafından geliştirilmektedir ve çoğu zaman benzer ya da aynı işi yapmak üzere birçok seçenek bulunmaktadır. Her Java kütüphanesinde birçok sınıf bulunduğu düşünülürse, bir uygulama geliştirilirken aynı adlı birçok sınıf ile karşılaşmanın olası olduğu görülür. Örneğin eğer İngilizce dilini kullanarak kod geliştiriyorsak ve tarih bilgisini yöneten bir sınıf yazarsak, sınıfın adını büyük olasılıkla **`Date`** olarak seçeriz. Java API de **`Date`** isimli bir sınıf

içermektedir. Peki bu durumda hangi **Date** sınıfının bizim yazdığımız sınıf, hangisinin Java API'deki sınıf olduğunun ayrımını nasıl yapacağız? Daha genel bir ifadeyle, birçok insan kod geliştirir ve büyük olasılıkla aynı sınıf adlarını kullanır durumdayken, bu sınıflar birbirlerine karıştırmadan nasıl kullanılabilir?

İşte burada paket kavramı devreye giriyor. Paket, her sınıfın adının biricik (tek) olmasını sağlayan bir programlama aracıdır. Farklı paketlerde bulunan sınıflar, adları aynı olsa bile birbirlerinden paket adları kullanılarak ayrılırlar. Örneğin Java API'deki **Date** sınıfı **java.util** adlı bir pakette bulunmaktadır ve tam adı **java.util.Date**'dir. Bizim yazacağımız sınıfın paket adını farklı seçmemiz durumda, aynı adı kullandığı halde diğer sınıf ile karışmayan bir sınıf kodlamış oluruz.

Paketler, disk üzerindeki klasörlere karşılık gelirler ve içiçe klasör yapısı oluşturacak şekilde paket adları tanımlanabilir. Yani örneğin **Zaman** sınıfını **kitap.java.ornekler.zamanornegi** adlı bir paketin içine koymak için, aşağıdaki klasör yapısının oluşturulması ve **Zaman** sınıfının **zamanornegi** adlı klasörün içinde bulunması gerekir:



Şekil 5-4. Paket ve klasör yapısı

Bir sınıfın paket adını belirlemek için ise, sınıf tanımının başına paket adının yazılması gerekir. Bunun için **package** anahtar sözcüğü, kendisinden sonra paketin adı gelecek şekilde kullanılır:

package ***paketadi;***

Aşağıda, **Zaman** sınıfının **kitap.java.ornekler.zamanornegi** paketine nasıl koyulduğu görülmektedir:

```
package kitap.java.ornekler.zamanornegi;
```

```
public class Zaman {
....
}
```

Kod 5-21. Paket tanımlama

> ***İsimlendirme geleneği:***
>
> ***Paket adı verilirken, paket adında bulunan herhangi bir kısım birden fazla sözcük içeriyorsa, sözcükler bitişik ve tamamı küçük harfle yazılır!***
>
> ***Örnek: kitap.java.ornekler.zamanornegi adlı pakette, zamanornegi kısmı zamanOrnegi ya da zaman_ornegi şeklinde değil zamanornegi şeklinde yazılmıştır.***

Farklı paketlerdeki sınıflara erişme:

Bir paketin içinde bulunan sınıfın kullanılabilmesi için sınıfın tam adının (paket adı ile birlikte sınıf adının) kullanılması gerekir. Ancak aynı paket içinde bulunan sınıflar birbirlerine paket adı kullanmadan da erişebilirler. Örneğin, **Program** sınıfını da **Zaman** sınıfı ile aynı pakete koyarsak, **Program** sınıfının içinde **Zaman** sınıfını kullanmak için paket adını kullanmamız gerekmez:

```
package kitap.java.ornekler.zamanornegi;

public class Program {
      public static void main(String[] args) {
            Zaman zaman1 = new Zaman();
            ....
      }
}
```

Kod 5-22. Aynı paketteki sınıfa erişim

Eğer **Program** sınıfı **Zaman** sınıfı ile ayne pakette değilse, **Zaman** sınıfına erişebilmek için iki yol vardır:

1- Zaman sınıfının tam adı kullanılır:

```
package kitap.java.ornekler;
public class Program {
```

```
      public static void main(String[] args) {
            kitap.java.ornekler.zamanornegi.Zaman zaman1 =
                   new kitap.java.ornekler.zamanornegi.Zaman();
            .....
      }
}
```

Kod 5-23. Farklı paketteki sınıfa erişim - 1

2 – **import** deyimini kullanılır: **import** deyimini kullanan sınıf içinde, başka bir paketteki sınıf(lar) kullanılabilir. **import**, JVM'ye, o sınıf içinde adı geçen sınıfların hangi paketlerde aranması gerektiğini bildiren bir deyimdir:

```
package kitap.java.ornekler;
import kitap.java.ornekler.zamanornegi.Zaman;
public class Program {
      public static void main(String[] args) {
            Zaman zaman1 = new Zaman();
            ....
      }
}
```

Kod 5-24. Farklı paketteki sınıfa erişim - 2

Görüldüğü gibi **import** deyimi kullanıldığında **Zaman** sınıfının tam adının kullanılması gerekmez. Dolayısıyla eğer **Zaman** sınıfı kodun çok yerinde geçiyorsa, her seferinde tam adı yazmamak için sınıf tanımının üstüne **import** deyimi yazılarak kodlama kolaylaştırılır.

Bir paketin içindeki sınıflardan birden fazlası kullanılmak isteniyorsa,

```
import paket1.sinif1;
import paket1.sinif2;
import paket1.sinif3;
....
```

şeklinde birden fazla satır yazılabilir. Ancak aynı paket içindeki birçok sınıfı tek tek yazmak yerine,

```
import paket1.*;
```

yazılarak o paketteki bütün sınıfların bu sınıfın kodu içinde kullanılması sağlanabilir. Böylece tek bir deyimle, birçok satır kodlamaktan kurtulmak olanaklıdır.

`Paket adlarının verilmesi:`

Biricik sınıf adlarının oluşturulabilmesi için paket adlarının da biricik olmaları gerekmektedir. Bunun sağlanabilmesi için genelde her kurum kendi alan adını (web adresinde kullanılan ad) kullanır. Çünkü alan adlarının dünyada biricik oldukları garanti olduğuna göre, alan adları kullanılarak oluşturulan paket adları da biricik olacaktır. Örneğin Hacettepe Üniversitesi Bilgisayar Mühendisiliği Bölümünün alan adı cs.hacettepe.edu.tr'dir ve bu ad dünya genelinde biriciktir.

Paket adı oluşturulurken alan adı ters çevrilerek yazılır. Böylece aynı kurumun oluşturduğu bütün paketler aynı hiyerarşide saklanmış olur. Örneğin, H.Ü. Bilgisayar Mühendisliği Bölümü'nün Bil132 dersinde geliştirilen 1. projenin altındaki bütün paketler,

`tr.edu.hacettepe.cs.bil132.proje1`

adlı bir paketin altında toplanabilir. Daha alt düzeydeki adlandırma ise sınıflar arasındaki ilişkiler ve modelin ayrıntılarına bağlı olarak yapılabilir.

`Hangi sınıflar hangi paketlere:`

Bir sınıfın hangi pakete koyulacağına programcı ya da o programcının çalıştığı kurumun standatları karar verir. Ancak genel eğilim, birbiri ile yakın ilişki içinde bulunan sınıfların bir arada toplanmasını sağlayacak şekilde aynı pakete koyulmaları, pakete de o paketteki sınıfların bir arada hangi işi yaptıklarını belirten bir isim vermek yönündedir.

5.3.3 Erişim Düzenleyiciler

Sarmalama ilkesi bir sınıf içindeki niteliklere o sınıf dışından doğrudan erişilmesinin engellenerek, erişimin o sınıfın içine kodlanmış yöntemler çağrılarak gerçekleştirilmesini söylemektedir. Öyleyse, bir sınıfın belirlenen öğelerine erişimi düzenleyecek programlama dili araçlarına gereksinim vardır.

Erişim düzenleyiciler, önüne yazıldıkları nitelik, yöntem ya da sınıfın erişilebilecekleri alanı belirleyen ayrılmış sözcüklerdir.

`private`: Önüne yazıldığı ögenin yalnızca o sınıf içinde doğrudan erişilebilir, o sınıfın dışındaki kod kesimlerinden doğrudan erişilemez olmasını sağlar. Sarmalama ilkesi gereği, niteliklerin yalnızca sınıf içinden doğrudan erişilebilir olmaları gerekmektedir. Bu nedenle genel olarak nitelikler **`private`** erişim düzenleyicisi ile tanımlanır. Bunun yanısıra, yalnızca sınıf içinden çağrılabilir olması istenen (örneğin sınıfa özel bazı hesaplamaları yapmak için kodlanan) yöntemler de **`private`** olarak tanımlanabilir.

`public`: Önüne yazıldığı ögenin yalnızca o sınıf içinde değil, o sınıfın dışında kalan bütün kod kesimlerinden de doğrudan erişilebilir olmasını sağlar. Sınıfa ait nesnelerin, diğer nesneler tarafından çağrılabilmesi istenen yöntemleri **`public`** erişim düzenleyicisi ile tanımlanır.

`protected`: **`private`** ve **`public`** erişim düzenleyicilerinin arasında bir erişim alanı tanımlar. **`protected`** tanımlanan öge, kendisi ile aynı pakette bulunan kod kesimlerinden doğrudan erişilebilir. Ayrıca kalıtım konusu anlatılırken belirtileceği gibi **`protected`** tanımlanmış ögeler, alt sınıftan da doğrudan erişilebilen ögelerdir.

Eğer bir programlama ögesinin önüne herhangi bir erişim düzenleyici yazılmazsa, o ögenin erişimi içinde bulunduğu paketle sınırlandırılmıştır. Yani aynı pakette bulunan sınıfların içinden o ögeye doğrudan erişilebilirken, başka paketlerden erişilemez (**`Zaman`** ve **`Program`** sınıflarına erişim düzenleyicileri yazmadan önce, bütün nitelik ve yöntemler paket erişim kuralları ile çalışıyordu. Hepsi aynı pakette olduğu için yazdığımız kodlar sorunsuz çalışıyordu).

5.3.4 get/set Yöntemleri

Sarmalama ilkesi gereği sınıfın niteliklerini **`private`** tanımladık. Böylece sınıfın dışında kalan herhangi bir kod kesiminde bu niteliklerin doğrudan

kullanılmasını engellemiş olduk. Ancak bu engelleme aynı zamanda niteliklerin değerlerinin kurulmasını ve sorgulanmasını da engeller.

Örneğin, şu ana kadar **`Zaman`** sınıfımızda kurucular, **`ilerle`** yöntemleri ve **`zamaniYaz`** yöntemi tanımlı. Fakat herhangi bir anda başka bir nesne tarafından **`saat'`**in (tek başına **`saat`** niteliğinin) kaç olduğunun sorulabilmesi ya da **`Zaman`** nesnesi oluşturulduktan sonra zamanın (örneğin 16:54:23 anına) ayarlanması olanaklı değil. Başka bir deyişle, zamanı ayarlamak ya da öğrenmek için (**`zamaniYaz`** yöntemi yalnızca zamanı ekrana yazdırıyor, başka bir nesnenin zamanı öğrenmesini sağlamıyor) başka yöntemlere gereksinimimiz var.

Java'da niteliklerin değerlerini sorgulayan yöntemlere genel olarak **`get`** yöntemleri, niteliklerin değerlerini kuran yöntemlere **`set`** yöntemleri (*getter and setter methods, accessors*) adı verilir.

`get ve set yöntemlerinin adlandırılması`: **`get`** ya da **`set`** sözcüğünden hemen sonra değeri sorgulanacak ya da kurulacak niteliğin adı ilk harfi büyütülerek yazılır. Örneğin saat niteliğinin değerini sorgulayan **`get`** yöntemi **`getSaat`**, değerini kuracak **`set`** yöntemi ise **`setSaat`** şeklinde adlandırılır.

`get/set` yöntemlerinin bütün nitelikler için yazılması zorunlu değildir. Örneğin **`Zaman`** sınıfında, eğer nesne bir kez oluşturulduktan sonra zamanın yalnızca ilerletilebilmesini istiyor, niteliklerin değerlerinin başka hiçbir şekilde değişmemesini istiyorsak, **`Zaman`** sınıfına **`set`** yöntemi yazmayız. Benzer şekilde eğer herhangi bir niteliğin değerinin başka bir nesne tarafından öğrenilmesini istemiyorsak **`get`** yöntemi yazmayabiliriz. Hangi yöntemlerin yazılacağı programcının tasarımına bağlıdır.

Biz şimdi **`Zaman`** sınıfımız için erişim düzenleyicileri de kullanarak **`get`** ve **`set`** yöntemlerini örnekleyelim:

```
public class Zaman {


      private int saat;
```

```java
	private int dakika;
	private int saniye;

	public Zaman() {
		saat = 0;
		dakika = 0;
		saniye = 0;
	}

	public Zaman(int st, int dk, int sn) {
		setSaat(st);
		setDakika(dk);
		setSaniye(sn);
	}

	public void ilerle() {
		saniye++;
	}

	public void ilerle(int sn) {
		saniye += sn;
		if (saniye > 59) {
			dakika += saniye / 60;
			saniye %= 60;
			if (dakika > 59) {
			saat += dakika / 60;
				dakika %= 60;
				if (saat > 23) {
					saat %= 24;
				}
			}
		}
	}

	public void zamaniYaz() {
		System.out.println("Şu anki zaman: " +
				saat + ":" + dakika + ":" +   (saniye % 60));
	}

	public int getSaat() {
```

```
		return saat;
	}

	public void setSaat(int st) {
		if (st >= 0 && st < 60) {
			saat = st;
		}
	}

	public int getDakika() {
		return dakika;
	}

	public void setDakika(int dk) {
		if (dk >= 0 && dk < 60) {
			dakika = dk;
		}
	}

	public int getSaniye() {
		return saniye;
	}

	public void setSaniye(int sn) {
		if (sn >= 0 && sn < 60) {
			saniye = sn;
		}
	}
}
```

Kod 5-25. Erişim düzenleyiciler ve get/set yöntemleri ile Zaman sınıfı

Dikkat ederseniz **`Zaman`** sınıfının bütün niteliklerini **`private`**, yöntemlerini ise **`public`** tanımladık. Kurucuların **`public`** olması, bu kurucuların başka sınıflar içinde kullanılarak **`Zaman`** sınıfına ait nesnelerin oluşturulabilmesini sağlar. Örneğin eğer parametre almayan **`Zaman()`** kurucusunu **`private`** tanımlarsak, **`Program`** sınıfının içindeki **`Zaman zaman1 = new Zaman();`** satırı derleme hatası alacaktır. Çünkü **`Program`** sınıfı **`Zaman`** sınıfının dışında

kalan bir kod kesimidir ve parametre almayan kurucu **`private`** tanımlanmıştır.

Kod 7-15'te **`Zaman`** sınıfının kendisi de **`public`** tanımlanmıştır. Hatırlarsanız bir dosyayı, içine yazdığımız sınıfın adı ile kaydediyorduk. Bir dosyanın içinde yalnızca bir sınıf bulunması gibi bir zorunluluk bulunmamakla birlikte en fazla bir tane **`public`** tanımlanmış sınıf bulunması zorunluluğu vardır. Bu şu anlama gelir: Bir dosyanın içine yalnızca bir tanesi **`public`** olmak şartı ile birden fazla sınıf tanımlanabilir. Diğer sınıfların önüne herhangi bir erişim düzenleyici yazılmaz (*package access*). Dosya da **`public`** olan sınıfın adı ile kaydedilmek zorundadır. **`public`** olmayan sınıflar ise ancak o sınıflarla aynı pakette olan sınıflar tarafından kullanılabilir.

Programlama dillerinin çeşitli yetenekleri vardır. Bu yeteneklerin var olmaları, her uygulamada kullanılmaları gerektiği anlamına gelmez. Geliştireceğiniz Java programlarında çok özel durumlar dışında bir dosyanın içine birden fazla sınıf yazmanız gerekmeyecektir. Dolayısıyla bu yeteneği hiçbir zaman kullanmayabilirsiniz de! Bir dosyanın içinde tek bir sınıf bulunması ve o sınıfın da *`public`* ***tanımlanması, uygulamalarınızın daha kolay yönetilebilir olmasını sağlayacaktır.***

5.4 Sınıf/Nesne Değişkenleri ve Yöntemleri

Şu ana kadar, sınıfların sadece tanımlar olduğunu, **`new`** ile bir nesne oluşturulana kadar bellekte fiziksel olarak yer ayrılmadığını söyledik. Bir nesne oluşturulduktan sonra, sınıfta tanımı yapılan her nitelik için o nesneye özel bir kopya oluşturulduğunu, o nesnenin içindeki herbir nitelik için ayrı ayrı yer alındığını gördük. Bu durumda, eğer iki **`Zaman`** nesnemiz varsa, her nesnenin içinde birer tane olmak üzere toplamda ikişer tane, birbirinden bağımsız bellek alanlarında saklanan **`saat`**, **`dakika`** ve **`saniye`** niteliklerimiz vardır.

Her nesne, ait olduğu sınıfa ait nitelikler için kendi içinde birer kopya bulundurmaktadır. Bu niteliklere **olgu değişkenleri (*instance variables*)** de denir. Olgu, nesneye karşılık gelir. Sınıfın bir olgusunun oluşturulması bir nesne oluşturulması anlamına gelmektedir. Her nesne için ayrı ayrı oluşturulan niteliklere de o olgunun değişkenleri adı verilir.

Benzer şekilde, sınıf içinde tanımladığımız yöntemleri nesne referansları üzerinden çağırırız. Oluşturulan nesnenin adresini tutan referans değişkeni kullanılarak yöntemin adı verilir ve o yöntem, hangi nesne üzerinden çağrılmışsa o nesnenin nitelikleriyle, başka bir deyişle o olgunun değişkenleri ile çalışır. Böylece iki farklı **`Zaman`** nesnesine ait referanslar üzerinden **`zamaniGoster()`** yöntemini çağırdığımızda ekranda iki farklı zaman yazdığını görürüz:

```
public static void main(String[] args) {
      Zaman zaman1 = new Zaman();
      Zaman zaman2 = new Zaman(14, 12, 34);

      zaman1.ilerle(12136);
      zaman1.zamaniYaz();

      zaman2.ilerle();
      zaman2.zamaniYaz();
}
```

Kod 5-26. Olgu değişkenleri ve olgu yöntemleri

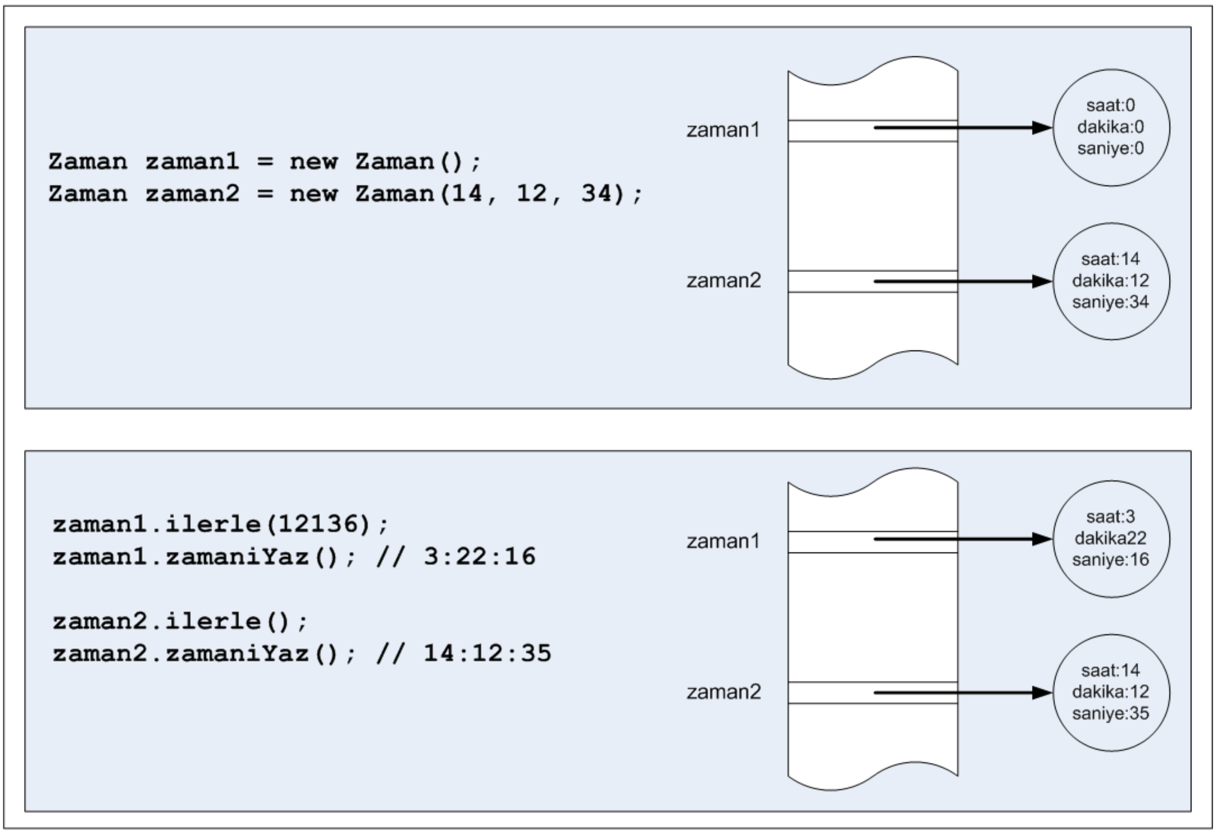
Yukarıdaki kod kesiminde **`Zaman`** sınıfına ait iki nesne oluşturuluyor ve bu nesnelerin referansları **`zaman1`** ve **`zaman2`** adlı iki referans tipinde değişkende saklanıyor. **`zaman1`** nesnesi üzerinden **`ilerle()`** yöntemi **`12136`** parametresi ile çağrıldığında, **`zaman1`** olgusuna ait değişkenlerin değerleri değişiyor. **`zaman1.zamaniYaz()`** yöntemi ile bu değişkenlerin değerleri ekrana yazdırılıyor.

`zaman2` olgusu üzerinden ise parametresiz **`ilerle()`** yöntemi çağrılıyor. Bu yöntem, üzerinden çağrıldığı referans ile gösterilen olguya ait olan değişkenler üzerinde işlem yapıyor. **`zaman2.zamaniYaz()`** yöntemi, bu olgunun değişkenlerinin değerlerini yazıyor. Böylece çıktıda iki farklı zaman değeri görüyoruz:

```
Şu anki zaman: 3:22:16
Şu anki zaman: 14:12:35
```

Çıktı 5-3. Olgu değişkenleri ve olgu yöntemleri

Bu durum aşağıdaki şekille daha net anlaşılabilir:



Şekil 5-5. Olgu değişkenleri ve olgu yöntemleri

Olgu değişkenlerinden başka, herhangi bir olguya değil, sınıfa ait olan değişkenler de tanımlanabilir. Bunlara sınıf değişkenleri denir. Aynı şekilde, bir nesne referansı yerine, sınıf adı üzerinden çağrılabilen yöntemler de yazılabilir, bunlara da sınıf yöntemleri adı verilir.

5.4.1 `static` anahtar sözcüğü

`static` anahtar sözcüğü, sınıf değişkenlerini ya da sınıf yöntemlerini tanımlamak için kullanılır.

Eğer bir sınıfın niteliklerinden bir ya da birkaçının önüne **`static`** yazılırsa, o nitelik(ler) sınıf değişkeni olur. Sınıf değişkenlerinin iki önemli özelliği vardır:

1. Sınıf değişkeni olarak tanımlanan nitelikler, her nesne için ayrı ayrı oluşturulmazlar. Sınıfa ait kaç nesne olursa olsun sınıf değişkeni 1 tanedir. Hangi nesne üzerinden erişilirse erişilsin bu değişkene erişilecektir.
2. Sınıf değişkenleri, hiç nesne oluşturulmasa da bellekte yer kaplarlar.

Eğer bir sınıfın yöntemlerinden bir ya da birkaçının önüne **`static`** yazılırsa, o yöntem(ler) sınıf yöntemi olur. Sınıf yöntemlerinin önemli bir özelliği vardır:

- Sınıf yönteminin çağrılabilmesi için o sınıfa ait nesne oluşturulması gerekmez. Sınıf yöntemine sınıf adı üzerinden ulaşılabilir.

Şimdi bu bilgileri örnekleyen basit bir kod yazalım:

```
public class A {
	private int olguDegiskeni;
	private static int sinifDegiskeni;

	public int getOlguDegiskeni() {
		return olguDegiskeni;
	}
	public void olguDegiskenineEkle(int sayi) {
		olguDegiskeni += sayi;
	}
```

```
	public static void sinifDegiskenineEkle(int sayi) {
		sinifDegiskeni += sayi;
	}
	public static int getSinifDegiskeni() {
		return sinifDegiskeni;
	}
}
```

Kod 5-27. static anahtar sözcüğü örneği – A sınıfı

```
public class Program {
	public static void main(String[] args) {
		A a1 = new A();
		a1.olguDegiskenineEkle(5);
		System.out.println("a1 - Olgu değişkeni: " +
					a1.getOlguDegiskeni());

		A.sinifDegiskenineEkle(7);
		System.out.println("a1 - Sınıf değişkeni: " +
					a1.getSinifDegiskeni());
		System.out.println("A - Sınıf değişkeni: " +
					A.getSinifDegiskeni());

		A a2 = new A();
		System.out.println("a2 - Sınıf değişkeni: " +
					a2.getSinifDegiskeni());
		a2.olguDegiskenineEkle(15);
		System.out.println("a2 - Olgu değişkeni: " +
					a2.getOlguDegiskeni());

		A.sinifDegiskenineEkle(27);
		System.out.println("a2 - Sınıf değişkeni: " +
					a2.getSinifDegiskeni());
		System.out.println("A - Sınıf değişkeni: " +
					A.getSinifDegiskeni());
	}
}
```

Kod 5-28. static anahtar sözcüğü örneği – Program sınıfı

```
a1 - Olgu değişkeni: 5
a1 - Sınıf değişkeni: 7
A - Sınıf değişkeni: 7
```

```
a2 - Sınıf değişkeni: 7
a2 - Olgu değişkeni: 15
a2 - Sınıf değişkeni: 34
A - Sınıf değişkeni: 34
```

Çıktı 5-4. static anahtar sözcüğü örneği – Çıktı

`A` sınıfının koduna bakalım: bir olgu değişkeni, bir de sınıf değişkeni tanımlanmış:

```
	private int olguDegiskeni;
	private static int sinifDegiskeni;
```

Bu değişkenlere verilen sayıyı ekleyen iki yöntem tanımlanmış:

```
	public void olguDegiskenineEkle(int sayi) {
		olguDegiskeni += sayi;
	}
	public static void sinifDegiskenineEkle(int sayi) {
		sinifDegiskeni += sayi;
	}
```

Bu değişkenlerin değerlerini sorgulayan iki yöntem tanımlanmış:

```
	public int getOlguDegiskeni() {
		return olguDegiskeni;
	}
	public static int getSinifDegiskeni() {
		return sinifDegiskeni;
	}
```

Dikkat edilirse, sınıf değişkeninin değerini döndüren **`getSinifDegiskeni()`** adlı yöntemin **`static`** tanımlandığı görülür. **Sınıf değişkenleri üzerinde işlem yapan yöntemlerin sınıf yöntemi olarak (`static`) tanımlanmaları zorunludur.** Bunu şöyle bir mantıksal ilişkiyle açıklamaya çalışalım:

Bir sınıf değişkeninin, henüz nesne oluşturulmasa da bellekte fiziksel olarak yer kapladığını söylemiştik. Bir sınıf yönteminin ise nesne oluşturulmadan, sınıf adı üzerinden çağrılabildiğini belirttik. Bu durumda, nesne var olmadan çağrılabilecek olan sınıf yöntemlerinin, nesne var olmadan bellekte var olamayan olgu değişkenlerine erişmesi olanaklı değildir. Benzer şekilde, nesne var olmadan bellekte var olan sınıf

değişkenleri üzerinde işlem yapan yöntemlerin, nesne var olmadan çağrılabilmeleri gerekir. Dolayısıyla, sınıf değişkenleri üzerinde işlem yapan yöntemler sınıf yöntemleri olmalıdır.

Öte yandan, bir olgu yönteminin sınıf değişkenine erişmesi olanaklıdır. Çünkü olgu yöntemi nesne oluştuktan sonra çağrılacaktır. Daha nesne oluşmadan fiziksel olarak bellekte var olan sınıf değişkenine nesne yöntemi ile de erişilebileceği açıktır. Bununla birlikte, sınıf değişkenlerine olgu yöntemlerinden erişilmesi terecih edilen bir durum değildir.

Örnek koda geri dönelim. **`Program`** sınıfına baktığımızda **`A`** sınıfından iki ayrı nesne oluşturulduğunu görüyoruz. İlk nesne, **`a1`** adlı referans değişkenle gösteriliyor.

```
a1.olguDegiskenineEkle(5);
```

yöntem çağrısı ile, **`a1`** nesnesinin **`olguDegiskeni`** adlı niteliğine 5 değeri ekleniyor. Çıktının ilk satırında bu görülüyor:

```
a1 - Olgu değişkeni: 5
```

Daha sonra **`A`** sınıfının sınıf yöntemi çağrılıyor:

```
A.sinifDegiskenineEkle(7);
```

Sınıf yöntemi, **`sınıfDegiskeni`** adlı değişkenin değerine 7 ekliyor. Burada sınıf yönteminin sınıf adı ile çağrılışına dikkat edelim. Hemen sonraki satırda

```
System.out.println("a1 - Sınıf değişkeni: " + a1.getSinifDegiskeni());
```

çağrısı ile olgu üzerinden sınıf değişkeninin değerine erişiliyor. Bunun olanaklı olduğunu ama tercih edilmediğini söylemiştik. Bu çağrının sonucunda çıktıda sınıf değişkeninin değeri görülüyor:

```
a1 - Sınıf değişkeni: 7
```

Hemen sonraki satırda ise sınıf adı üzerinden sınıf yöntemi kullanılarak sınıf değişkeninin değeri sorgulanıp ekrana yazılıyor:

```
System.out.println("A - Sınıf değişkeni: " + A.getSinifDegiskeni());
```

Çıktıda oluşan değer ise olgu değişkenininden erişimde görülen değerle aynı:

```
A - Sınıf değişkeni: 7
```

Sonraki iki satırda **a2** nesnesi oluşturulup, **a2** olgusu ile ilgili hiçbir işlem yapılmadan sınıf değişkeninin değeri sorgulanıyor:

```
A a2 = new A();
System.out.println("a2 - Sınıf değişkeni: " + a2.getSinifDegiskeni());
```

Çıktıda ise şu satır oluşuyor:

```
a2 - Sınıf değişkeni: 7
```

a2 olgusu üzerinden hiçbir işlem yapılmadığı halde sınıf değişkeninin değeri önceki çıktı satırlarında görülenle aynı. Bunun nedeni, bütün olguların aynı sınıf değişkenine ulaşıyor olmaları. Sınıf değişkeninin bellekte sadece tek kopyası bulunuyor ve bütün olgular aynı değeri paylaşıyorlar. Dolayısıyla herhangi bir olgu üzerinden ya da sınıf adı üzerinden sınıf değişkeninin değeri değiştirilirse bütün olgular aynı değere ulaşıyorlar.

Sınıf değişkenleri ne için kullanılır?

Çok kullanıcılı bir oyun programı yaptığımızı düşünelim. Her oyuncu kendi ekranında o andaki oyuncuların sayısını görmek istiyor olsun. Bunu yapabilmek için **Oyuncu** sınıfının her nesnesinin, **Oyuncu** sınıfına ait nesnelerin sayısına ulaşabilmesi gerekir. Eğer oyuncu sayısını bir olgu değişkeni olarak tanımlarsak, her olgunun kendine ait birer değişkeni olacağından, yeni bir **Oyuncu** nesnesi oluşturulduğunda ya da bir oyuncu oyundan çıktığında, her olgudaki oyuncu sayısı değişkenini güncellememiz gerekecektir. Sınıf değişkeninin tek kopyası olduğundan, oyuncu sayısını sınıf değişkeni olarak tanımlarsak güncellenmesi gereken tek değer sınıf değişkeni olur. Bütün olgular sınıf değişkenine erişebileceği için tek ve merkezi bir noktadan bu bilgiyi yöntemiş oluruz.

Kısaca, sınıf değişkenleri **sınıfın bütün nesneleri için genel değişken** (*global variable*) olarak düşünülebilirler. Eğer verinin tek kopyasına gereksinim varsa ya da bütün nesnelerce paylaşılması gereken veriler varsa, bu tip verilerin sınıf değişkeni olarak tanımlanmaları uygundur.

Aşağıda **Oyuncu** sınıfının kodu görülmektedir:

```
public class Oyuncu {
      private static int oyuncuSayisi;
      public Oyuncu() {
            Oyuncu.oyuncuSayisi++;
      }
      public void oyundanCik() {
            Oyuncu.oyuncuSayisi--;
      }
      public static int kacOyuncuVar() {
            return Oyuncu.oyuncuSayisi;
      }
}
```

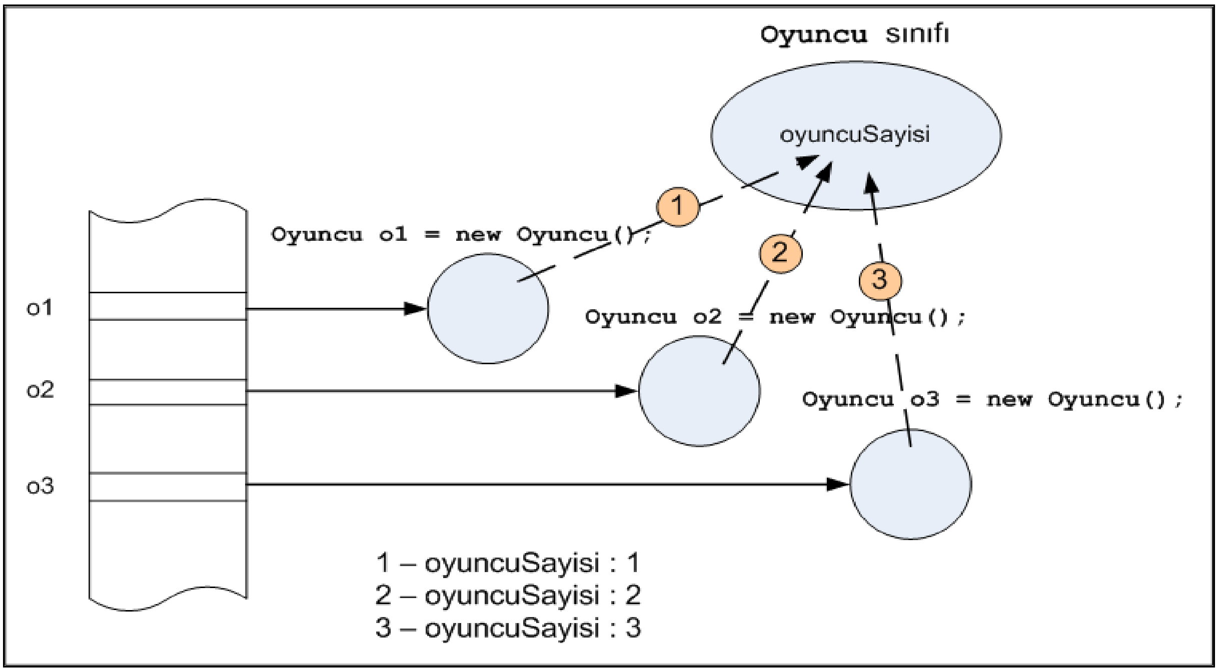
Kod 5-29. Oyunu sınıfı

Aşağıda `Oyuncu` sınıfını kullanan örnek bir program görülmektedir:

```
public class Program {
      public static void main(String[] args) {
            System.out.println("Oyuncu sayısı: " + Oyuncu.kacOyuncuVar());
            Oyuncu o1 = new Oyuncu();
            Oyuncu o2 = new Oyuncu();
            Oyuncu o3 = new Oyuncu();
            System.out.println("Oyuncu sayısı: " + o1.kacOyuncuVar());
            o1.oyundanCik();
            o3.oyundanCik();
            System.out.println("Oyuncu sayısı: " + o2.kacOyuncuVar());
      }
}
```

Kod 5-30. Oyuncu sınıfını kullanan örnek program

Aşağıdaki şekilde her Oyuncu nesnesi oluşturulduğunda sınıf değişkeninin değerinin nasıl değiştiği görülmektedir:



Şekil 5-6. Sınıf değişkeninin olgulardan güncellenmesi

Sınıf yöntemleri ne için kullanılır?

Bazen bir yöntemi çağırmak için nesne oluşturmaya gerek yoktur. Örneğin karekök bulan bir yöntemin herhangi bir nesne üzerinden çağrılması çok anlamlı değildir. Çünkü karekök hesaplayabilmek için gereksinim duyulan tek bilgi hangi sayının karekökünün bulunacağıdır ve bu bilgi yönteme parametre olarak verilir. Oysa ortada bir nesne varsa, bu nesnenin bir de durumu vardır ve çağrılacak yöntem ancak bu durum bilgisinden faydalanacaksa nesnenin varlığı anlamlı olur. Karekök hesaplamak için herhangi bir durum bilgisine gereksinim olmadığına göre, bu yöntemi içine yazdığımız sınıfın (herşeyi bir sınıfın içine yazmak zorundayız) nesnesini oluşurmak anlamlı olmayacaktır. İşte böyle durumlarda, nesne oluşturulmadan çağrılabilmesini istediğimiz yöntemleri sınıf yöntemi olarak tanımlarız. Böylece nesne oluşturmadan yöntemi çağırmak olanaklı hale gelir.

Java API'de **java.lang.Math** sınıfında tam da bu işi yapan **sqrt** adlı yöntem şu şekilde tanımlanmıştır:

```
public static double sqrt(double a)
```

Java API belgelerinden **java.lang** paketi içindeki **Math** sınıfını incelerseniz, bu sınıfın bütün yöntemlerinin **static** tanımlandığını göreceksiniz. Bunun nedeni, burada tanımlanan yöntemlerin, yapacakları hesaplamalar için durum bilgisine gereksinim duymamalarıdır. Böylece karekök hesaplayıp ekrana yazdırmak için yapmamız gereken tek şey **Math** sınıfı üzerinden **sqrt** yöntemini çağırmak olacaktır:

```
System.out.println(Math.sqrt(25));
```

5.4.2 main yöntemi

Java programlarının çalışmaya **main** yönteminden başladığını daha önce söylemiştik. Ayrıca örneklerimizde de **main** yöntemini kodladık. Şimdi bu yönteme daha yakından bakalım.

```
public static void main(String[] args)  {
      ....
}
```

Kod 5-31. main yöntemi

Görüldüğü gibi, **main** yöntemi **public** tanımlanmıştır, yani sınıf dışından erişilebilir. Peki programın başladığı nokta **main** yönteminin kendisi ise, bu yönteme erişecek olan ve sınıfın dışında yazılmış olan kod kesimi hangisidir? Şu ana kadar yazdığımız örneklerin hiçbirisinde **main** yöntemini çağıran bir kod kesimi yazmadık.

main yöntemi **static** tanımlanmıştır. Demek ki, **main** yönteminin içine yazıldığı sınıfın bir nesnesi oluşturulmadan da bu yöntem çağrılabilir.

main yöntemi parametre olarak **String** tipinden nesnelerin referanslarını içeren bir dizi almaktadır. Peki bu parametre nereden gelmektedir?

Eğer yazdığımız her programın JVM tarafından işletileceğini hatırlarsak, bu sorular kolayca yanıtlanmış olur: **main** yöntemini JVM çağırır. JVM'nin bizim yazdığımız sınıfın içindeki yöntemi çağırabilmesi için bu yöntem **public** tanımlanmıştır. JVM'nin bizim yöntemimizi çağırmak için nesne oluşturması gerekmesin diye yöntem **static** tanımlanmıştır. **String** dizisini parametre olarak **main** yöntemine veren de gene JVM'dir. Bu

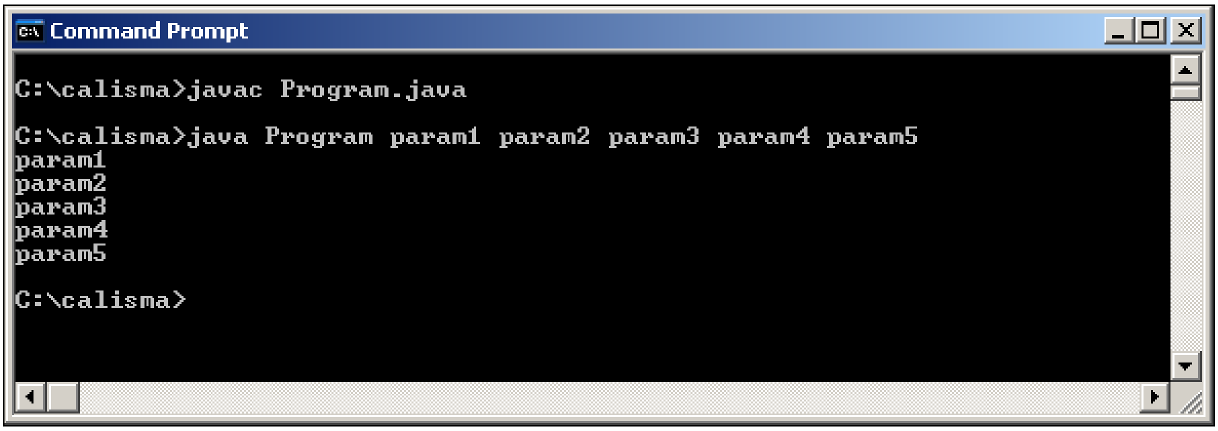
parametreyi programı çalıştırırken komut satırından (konsoldan) verdiğimiz değerlerden oluşturmaktadır.

Bu durumu aşağıdaki programla açıklayalım:

```
public class Program {
      public static void main(String[] args) {
            for (int i = 0; i < args.length; ++i) {
                  System.out.println(args[i]);
            }
      }
}
```

Kod 5-32. Komut satırından verilen parametreleri ekrana yazan program

Bu programı komut satırından parametre vererek çalıştıralım:



Şekil 5-7. Komut satırından parametre ile program çalıştırılması

Görüldüğü gibi programımız komut satırından verilen bütün parametreleri ekrana yazdı. Tabi ki gerçek bir uygulamada bu parametreler ekrana yazılmak yerine daha anlamlı işlemler yapmak üzere kullanılacaklardır.

5.4.3 `static` kod blokları

Olgu değişkenlerinin ilk değerlerinin verilmesi için kurucu yöntemleri kullanabiliriz. Ancak sınıf değişkenlerinin ilk değerlerini kurucu içinde vermeye çalışmak yanlış olabilir. Çünkü sınıf değişkenleri, hiç nesne oluşturulmamış olsa da kullanılabilmektedir. Sınıf değişkenleri ile ilgili olarak bir defaya özel olmak üzere işletilmesi istenen kod kesimleri **`static`** kod bloklarında kodlanabilir. **`static`** kod blokları, sınıf belleğe yüklendiği

anda işletilir. Böylece sınıf değişkenleri bellekte oluşturuldukları anda ilk değerlerini almış olurlar.

Bir **`static`** kod bloğu şu şekilde kodlanır:

```
static {
      deyim1;
      deyim2;
      ....
      deyimN;
}
```

Kod 5-33. static kod bloğunun yazılışı

Aşağıda, **`static`** kod bloğunun kullanılışını örnekleyen küçük bir program görülmektedir:

```
package yazarlar;

public class Yazar {
      private String ad;
      private String soyad;

      public Yazar(String ad, String soyad) {
            this.ad = ad;
            this.soyad = soyad;
      }
      public String getAd() {
            return ad;
      }
      public String getSoyad() {
            return soyad;
      }
      public String getBilgi() {
            return this.ad + " " + this.soyad;
      }
}
```

Kod 5-34. static kod bloğu örneği – Yazar.java

```
package yazarlar;

public class YazarIslemleri {
      private static Yazar[] yazarlar;
```

```
	static {
		yazarlar = new Yazar[5];
		yazarlar[0] = new Yazar("Reşat Nuri", "Güntekin");
		yazarlar[1] = new Yazar("Necip Fazıl", "Kısakürek");
		yazarlar[2] = new Yazar("Yakup Kadri", "Karaosmanoğlu");
		yazarlar[3] = new Yazar("Halit Ziya", "Uşaklıgil");
		yazarlar[4] = new Yazar("Yahya Kemal", "Beyatlı");
	}

	public static Yazar[] getYazarlar() {
		return YazarIslemleri.yazarlar;
	}
}
```

Kod 5-35. static kod bloğu örneği – YazarIslemleri.java

```
package yazarlar;

public class Program {

	public static void main(String[] args) {
		Yazar[] yazarlar = YazarIslemleri.getYazarlar();
		for (int i = 0; i < yazarlar.length; ++i) {
		    System.out.println(i + 1 + " - " + yazarlar[i].getBilgi());
		}
	}
}
```

Kod 5-36. static kod bloğu örneği – Program.java

Programın çıktısı:

```
1 - Reşat Nuri Güntekin
2 - Necip Fazıl Kısakürek
3 - Yakup Kadri Karaosmanoğlu
4 - Halit Ziya Uşaklıgil
5 - Yahya Kemal Beyatlı
```

Çıktı 5-5. static kod bloğu örneği – program çıktısı

5.4.4 `final` anahtar sözcüğü ve sabit tanımlama

`final`, önüne yazıldığı değişkenin değerinin bir kez verildikten sonra değiştirilemeyeceğini belirten bir anahtar sözcüktür. Dolayısıyla sabit tanımlamak için kullanılır. **`final`** anahtar sözcüğü ile bir sabit tanımı şu şekilde yapılabilir:

```
final double PI = 3.14;
```

Bir sabitin değeri verildikten sonra değiştirilemeyeceğine göre, sabit olarak tanımlanmış bir niteliğin **`public`** olmasında sarmalama ilkesi açısından herhangi bir sakınca yoktur. Çünkü sarmalama ilkesinin temel amacı, nesnenin durumunda değişiklik yapabilecek kod kesiminin yalnızca sınıfın içinde olmasını sağlayarak hem hatalara karşı korumalı, hem de kolay değiştirilebilir bir kod elde etmektir. Sabitler ise zaten değiştirilemeyeceklerinden **`public`** tanımlanmalarında ilkeye aykırı bir durum yoktur. Öyleyse **`PI`** sabitini **`public`** tanımlayalım:

```
public final double PI = 3.14;
```

Bir sabitin kullanılabilmesi için, sabitin içine tanımlandığı sınıfa ait bir nesne oluşturulması anlamsızdır. Çünkü sabitin nesnenin durumuna bağlılığı söz konusu değildir. Öyleyse sabite sınıf üzerinden erişilebilmesi gerekir. Bu durumda sabitleri static tanımlamak anlamlı olacaktır:

```
public final static double PI = 3.14;
```

Bir uygulamadaki sabitlerin bir arada tanımlanması düşünülebilir:

```
public class Sabitler {
	public final static int SISTEMDEKI_OGRENCI_SAYISI = 100;
	public final static int BIR_OGRENCININ_ALABILECEGI_DERS_SAYISI = 8;
	public final static int BIR_OGRENCININ_ALABILECEGI_KREDI_SAYISI = 22;
}
```

Kod 5-37. Sabitlerin tanımlandığı bir sınıf

> ***Sabit adları büyük harflerle yazılır. Sabit adı birden fazla sözcükten oluşuyorsa, sözcükler altçizgi (_) ile birbirlerinden ayrılır. Örneğin en fazla kayıt sayısını ifade edecek sabitin adı şöyle verilebilir: EN_FAZLA_KAYIT_SAYISI***

5.5 Kalıtım (*Inheritance*)

Sarmalama konusunu anlatırken, nesneye yönelik programlama yaklaşımının 3 ilkesi/aracı olduğundan ve bunlardan ilkinin sarmalama ilkesi olduğundan söz etmiştik.

Sarmalama ilkesi, niteliklere erişimi sınıfın içine kodlanan yöntemlerle sınırlandırarak kodun değiştirilebilirliğini arttıran ve bakımını kolaylaştıran ilkedir.

Kalıtım ise nesneye yönelik programlamanın bu 3 ilke/aracından ikincisidir. Kalıtımın getirisi, kodun yeniden kullanılabilirliğini sağlamasıdır. Buna ek olarak 3. araç olan çokbiçimlilik (*polymorphism*) de kalıtım sayesinde gerçekleştirilebilmektedir.

Bu bölümde önce nesneye yönelik bir modeldeki sınıflar arasında nasıl ilişkilerin bulunabileceğinden bahsedilecek ve kalıtımın gerçekleştirilebilmesi için modelde hangi ilişkinin bulunması gerektiği açıklanmaya çalışılacaktır. Daha sonra kalıtımın ne olduğu anlatılacak ve Java dilinde nasıl gerçekleştirildiği örneklerle gösterilecektir.

5.5.1 Nesneye Yönelik Bir Modelde Sınıflar Arası İlişkiler

Nesneye yönelik modeli oluşturmak için önce modeldeki sınıfların belirlenmesi gerekir. Sınıfların belirlenmesi için kullanılabilecek basit bir yöntem; gereksinimleri anlatan belgedeki varlık isimlerinin seçilmesi, sonra bu isimlerden hangilerinin sınıf, hangilerinin nitelik olabileceğine, sınıf olması düşünülenlerden hangilerinin başka sınıflarla nasıl ilişkileri olduğuna karar verilmesidir. Bu yaklaşım iyi bir başlangıç yapılmasını sağlayabilir. Daha sonra bu sınıfların üzerinden tekrar tekrar geçilerek gereksinimler irdelenir, gerekli yöntemler ve nitelikler belirlenir, sınıfların bazıları elenip, bazı yeni sınıflar eklenebilir.

Modelde hangi sınıfların yer alacağının belirlenmesi modelin oluşturulması için yeterli değildir. Bu sınıflar arasındaki ilişkilerin de şekillendirilmesi

gerekmektedir. Peki iki sınıf arasında bulunabilecek ilişkiler nelerdir ve bu ilişkiler nasıl ortaya çıkartılacaktır?

İki sınıf arasında bulunabilecek 3 temel ilişki vardır:

Bağımlılık (*Dependency*)/ Kullanır (“*uses a*”) ilişkisi: Eğer bir sınıfın yöntemlerinden en az bir tanesi başka bir sınıfa ait en az bir tane parametre alıyorsa, parametre alan yöntemin içinde bulunduğu sınıf diğer sınıfa ***bağımlıdır (depends)*** ya da onu ***kullanır*** (***uses***) denir. Aşağıdaki örnekle bu ilişkiyi açıklamaya çalışalım:

```
public class A {
      public void yontem(B b) {
            //....
      }
}
```

Kod 5-38. İki sınıf arasındaki bağımlılık ilişkisi

Bu kod kesimine göre “**A** sınıfı **B** sınıfına bağımlıdır” ya da “**A** sınıfı **B** sınıfını kullanır”. Bağımlılık ilişkisi, **A** sınıfını **B** sınıfının kodunda yapılabilecek değişikliklerde potansiyel olarak değişebilecek bir sınıf yaptığı için önemlidir. Eğer **A** sınıfı **B** sınıfını kullanıyorsa, **B** sınıfında yapılabilecek herhangi bir değişikliğin **A** sınıfını etkilemesi olasıdır. Dolayısıyla bu iki sınıf birlikte yönetilmelidir.

Geliştirilen bir uygulamada sınıflar arası bağımlılıkların artması demek, birlikte yönetilmesi gereken sınıf sayısının da artması demektir. Bu durumda kodun yönetilebilirliği ve bakım kolaylığı azalacaktır. Çünkü yapılacak her değişiklik potansiyel olarak başka sınıfların da değiştirilmesi sonucunu getireceğinden bakım yükü artmaktadır. Bunun önüne geçilebilmesi için uygulanan bir takım teknikler vardır. Tasarım Örüntüleri (*Design Patterns*) başlığı altında anlatılan tekniklerden bazıları sınıflar arası bağımlılıkların azaltılmasına yöneliktir. Burada tasarım örüntüleri anlatılmayacaktır ancak kabaca şu söylenebilir: sınıflar arası bağımlılıkların azaltılabilmesi için yapılabilecek en basit şey, birbiri ile yakın ilişki içinde bulunan sınıfları bir alt sistem gibi düşünüp, bu alt sistemin başka alt

sistemlerle ilişkisini kuracak tek bir sınıf kodlanmasıdır. Böylece bir alt sistemdeki sınıfların farklı alt sistemlerdeki sınıflarla bağımlılıkları azaltılabilir.

Birleştirme (*composition*) / İçerir ("*has a*") ilişkisi: Eğer bir sınıfın niteliklerinden en az bir tanesi başka bir sınıfın türündense, niteliğe sahip olan sınıf diğer sınıfı **içerir** denir. Aşağıdaki örnek içerme ilişkisini örneklemektedir:

```
public class A {
      private B b;
      // ....
}
```

Kod 5-39. İki sınıf arasındaki içerme ilişkisi

A sınıfının **B** sınıfına ait bir niteliği bulunduğuna göre **A** sınıfı **B** sınıfını **içerir** ("**A** ***has a*** **B**").

Kalıtım (*inheritance*) / "*is a*" ilişkisi: Eğer bir sınıfa ait bütün nesneler aynı zamanda daha genel bir başka sınıfa da aitse, o zaman bu iki sınıf arasında kalıtım ilişkisinden söz edilir. Örneğin her "Kara Taşıtı" bir "Taşıt", her "Motorlu Kara Taşıtı" aynı zamanda hem bir "Kara Taşıtı" hem de bir "Taşıt"tır. Dolayısıyla "Kara Taşıtı" ile "Taşıt" arasında bir kalıtım ilişkisi, "Motorlu Kara Taşıtı" ile "Kara Taşıtı" arasında da başka bir kalıtım ilişkisi bulunmaktadır.

Bazen kalıtım ilişkisi ile içerme ilişkisi birbirine karıştırılır. Oysa bu iki ilişki önemli bir farkla birbirlerinden ayrılırlar. Modeli oluştururken hangi ilişkinin doğru olduğunun bulunabilmesi için, aralarındaki ilişkinin bulunmaya çalışıldığı **A** ve **B** sınıfları için şu iki önermenin sınanması gerekir:

1. Önerme: "Her **A** bir **B**'dir (**A** *is a* **B**)": Kalıtım ilişkisi

2. Önerme: "Her **A**'nın bir **B**'si vardır (**A** *has a* **B**)": İçerme ilişkisi

Eğer "Her **A** bir **B**'dir" ifadesi doğru bir önerme ise **A** sınıfı ile **B** sınıfı arasında kalıtım ilişkisi vardır ve **A** sınıfı **B** sınıfından kalıtım yapmaktadır.

Eğer "Her **A**'nın bir **B**'si vardır" ifadesi doğru bir önerme ise **A** sınıfı ile **B** sınıfı arasında içerme ilişkisi vardır ve **A** sınıfının nitelikleri arasında **B** sınıfının türünden bir (ya da daha fazla) nitelik vardır.

Örneğin, **Nokta** sınıfı ile **Cember** sınıfı arasındaki ilişkinin nasıl bir ilişki olduğunu bulmaya çalışalım:

1. Önerme: "Her Cember bir Nokta'dır."

2. Önerme: "Her Cemberin bir merkez Nokta'sı vardır."

Görüldüğü gibi 2. önerme doğru bir önermedir. Öyleyse ilişki içerme ilişkisidir ve **Cember** sınıfının içinde **Nokta** sınıfına ait bir nitelik tanımlanması gerekmektedir.

İçerme ilişkisi de kalıtım ilişkisi de kodun yeniden kullanılabilirliğini sağlamaktadır. Ancak unutulmaması gereken nokta şudur: Nesneye yönelik model oluşturulurken, amaç kodun yeniden kullanılırlığını sağlamak değil doğru bir model oluşturmaktır. Öyleyse sınıflar arasındaki ilişkilerin doğru tanımlanması gerekir. Çünkü doğru oluşturulmuş bir model, zaten yanlış bir modele oranla daha kolay gerçekleştirilebilecek ve bakımı daha kolay yapılabilecek bir model olacaktır.

İki sınıf arasında birden fazla ilişki bulunuabilir. Örneğin **Cember** sınıfı ile **Nokta** sınıfı arasında içerme ilişkisi olduğu gibi, bağımlılık ilişkisi de olabilir: **Cember** sınıfının yöntemlerinden bir ya da daha fazlası **Nokta** sınıfından bir nesneyi parametre alabilir.

5.5.2 Java'da Kalıtım

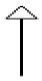
Kalıtım, programlama ortamında da gerçek hayattaki tanımına benzer bir işi gerçekleştirir. Bir sınıfın başka bir sınıftan kalıtım yapması demek, kalıtımı yapan sınıfın diğer sınıftaki nitelik ve yöntemleri kendisine alması demektir. Eğer kalıtımı yapan sınıfa **alt sınıf**, kendisinden kalıtım yapılan sınıfa **ata sınıf** dersek, ata sınıfta tanımlı olan herşeyin (kurucular hariç) alt sınıf için de tanımlı olduğunu söyleyebiliriz.

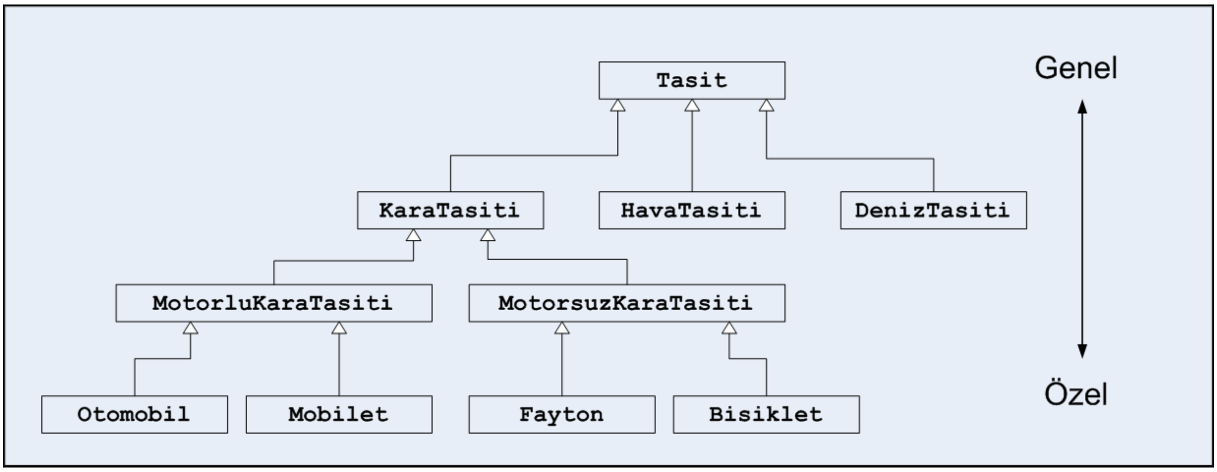
Java'da kalıtım, **`extends`** anahtar sözcüğü ile gerçekleştirilir. Eğer bir **`A`** sınıfın **`B`** sınıfından kalıtım yapması isteniyorsa, **`A`** sınıfı

```
public class A extends B {
}
```

şeklinde tanımlanır.

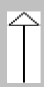
Bu tanımlama, **`B`** sınıfının içine kodlanmış bütün niteilk ve yöntemlerin (kurucular hariç herşeyin) **`A`** sınıfı için de tanımlı olmasını sağlar. Böylece **`B`** sınıfına yazılan kodlar **`A`** sınıfı içine yazılmadan **`A`** sınıfı için de kullanılabilir. **`A`** sınıfına **`B`** sınıfının alt sınıfı (*subclass*) ya da çocuğu (*child*), **`B`** sınıfına da **`A`** sınıfının ata sınıfı ya da atası (*ancestor*) denir.

Kalıtım ilişkisi, UML (*Unified Modeling Language*) dilinde  sembolü ile ifade edilir ve birden fazla düzeyli de kurulabilir. Aşağıda, **`Tasit`** sınıfı ve o sınıfın alt sınıflarından bazılarını gösteren bir kalıtım ağacı görülmektedir.



Şekil 5-8. Tasit sınıfının kalıtım ağacı

Kalıtım ağacında yukarıya doğru çıkıldıkça genele, aşağıya doğru inildikçe özele gidilir. `Tasit`, en genel ifade ile bütün taşıtları belirtir. `Bisiklet` ise, çok daha özel bir taşıttır. Ata sınıflar, alt sınıfların daha genel bir kümesidir.

Kalıtımı belirten  ***sembolü kendisinden kalıtılan sınıf yönünde izlenirse, genele doğru gidilmiş olur. Bu nedenle bu ilişkiye UML bağlamında genelleştirme (generalization) ilişkisi de denir.***

`Tasit` sınıfının ve alt sınıflarının örnek olarak gösterildiği yukarıdaki ağacı oluşturmak üzere yazılacak sınıf tanımları aşağıdaki tanımlara benzeyecektir:

```
public class Tasit {
}
public class KaraTasiti extends Tasit {
}
public class HavaTasiti extends Tasit {
}
public class DenizTasiti extends Tasit {
}
public class MotorluKaraTasiti extends KaraTasiti {
}
public class MotorsuzKaraTasiti extends KaraTasiti {
}
public class Otomobil extends MotorluKaraTasiti {
}
public class Mobilet extends MotorluKaraTasiti {
}
public class Fayton extends MotorsuzKaraTasiti {
}
public class Bisiklet extends MotorsuzKaraTasiti {
}
```

Kod 5-40. Tasit sınıfının kalıtım ağacını oluşturacak sınıf tanımları

Kalıtım ağacının derinliğini sınırlayan herhangi bir kural yoktur. Derinlik ne olursa olsun, ağacın alt yapraklarındaki sınıflar, kendi üst yapraklarındaki sınıfların alt sınıflarıdır. Örneğin **`Mobilet`** sınıfı hem **`MotorluKaraTasiti`** sınıfının alt sınıfı, hem **`KaraTasiti`** sınıfının alt sınıfı, hem de **`Tasit`** sınıfının alt sınıfıdır. Bununla birlikte, **`Mobilet`** sınıfı **`MotorluKaraTasiti`** sınıfının doğrudan alt sınıfı (*direct subclass*), **`KaraTasiti`** ve **`Tasit`** sınıflarının ise dolaylı alt sınıfıdır (*indirect subclass*). Alt sınıflar kendi üstlerindeki bütün sınıfların nitelik ve yöntemlerini kalıtım ile alır. Örnek kalıtım ağacımıza göre, **`KaraTasiti`** sınıfı kendi atası olan **`Tasit`** sınıfından, **`MotorluKaraTasiti`** sınıfı **`KaraTasiti`** sınıfından, **`Mobilet`** sınıfı da **`MotorluKaraTasiti`** sınıfından bütün nitelik ve yöntemleri alacaktır.

Böylece en üstteki **`Tasit`** sınıfında tanımladığımız bir yöntem, en alttaki **`Mobilet`** sınıfı için de tanımlı olacaktır.

Şimdi bu anlattıklarımızı kod ile görmeye çalışalım:

```
package kalitim;
public class Tasit {
      public void ilerle(int birim) {
            System.out.println("Taşıt " + birim + " birim ilerliyor..");
      }
}
```

Kod 5-41. Kalıtım örneği – Tasit sınıfı

```
package kalitim;
public class KaraTasiti extends Tasit {
      private int tekerlekSayisi;

      public int getTekerlekSayisi() {
            return this.tekerlekSayisi;
      }
}
```

Kod 5-42. Kalıtım örneği – KaraTasiti sınıfı

```
package kalitim;
public class HavaTasiti extends Tasit {
      // sınıf tanımı bu örnek için boş bırakılmıştır
}
```

Kod 5-43. Kalıtım örneği – HavaTasiti sınıfı

```
package kalitim;
public class DenizTasiti extends Tasit {
      // sınıf tanımı bu örnek için boş bırakılmıştır
}
```

Kod 5-44. Kalıtım örneği – DenizTasiti sınıfı

```
package kalitim;
public class MotorluKaraTasiti extends KaraTasiti {
      private int motorHacmi;

      public int getMotorHacmi() {
            return motorHacmi;
      }
```

```
}
```

Kod 5-45. Kalıtım örneği – MotorluKaraTasiti sınıfı

```
package kalitim;
public class MotorsuzKaraTasiti extends KaraTasiti {
      // sınıf tanımı bu örnek için boş bırakılmıştır
}
```

Kod 5-46. Kalıtım örneği – MotorsuzKaraTasiti sınıfı

```
package kalitim;
public class Mobilet extends MotorluKaraTasiti {
      // sınıf tanımı bu örnek için boş bırakılmıştır
}
```

Kod 5-47. Kalıtım örneği – Mobilet sınıfı

```
package kalitim;
public class Otomobil extends MotorluKaraTasiti {
      // sınıf tanımı bu örnek için boş bırakılmıştır
}
```

Kod 5-48. Kalıtım örneği – Otomobil sınıfı

```
package kalitim;
public class Fayton extends MotorsuzKaraTasiti {
      // sınıf tanımı bu örnek için boş bırakılmıştır
}
```

Kod 5-49. Kalıtım örneği – Fayton sınıfı

```
package kalitim;
public class Bisiklet extends MotorsuzKaraTasiti {
      // sınıf tanımı bu örnek için boş bırakılmıştır
}
```

Kod 5-50. Kalıtım örneği – Bisiklet sınıfı

Yukarıdaki sınıfların kodları incelenirse, **`Tasit`** sınıfında **`ilerle()`** yöntemi, **`KaraTasiti`** sınıfında **`tekerlekSayisi`** niteliği ile **`getTekerlekSayisi()`** yöntemi, **`MotorluKaraTasiti`** sınıfında ise **`motorHacmi`** niteliği ile **`getMotorHacmi()`** yöntemlerinin kodlandığı, diğer sınıfların içinin şimdilik boş bırakıldıkları görülür.

Kalıtımı anlatırken herhangi bir alt sınıf, ata sınıflarındaki herşeyi alır demiştik. Bu durumda örneğin **`Bisiklet`** sınıfının, **`KaraTasiti`** sınıfında

tanımlanmış olan **tekerlekSayisi** niteliği ile **getTekerlekSayisi()** yöntemini, **Tasit** sınıfında tanımlanmış olan **ilerle()** yöntemini almış olmasını bekliyoruz. Bunu sınamak üzere örnek kod yazalım:

```
package kalitim;
public class Program {
      public static void main(String[] args) {
            Bisiklet bisiklet = new Bisiklet();
            bisiklet.ilerle(5);
            System.out.print("Bisikletin " + bisiklet.getTekerlekSayisi() +
                              " tekerleği var.");
      }
}
```

Kod 5-51. Kalıtım örneği – Örnek program

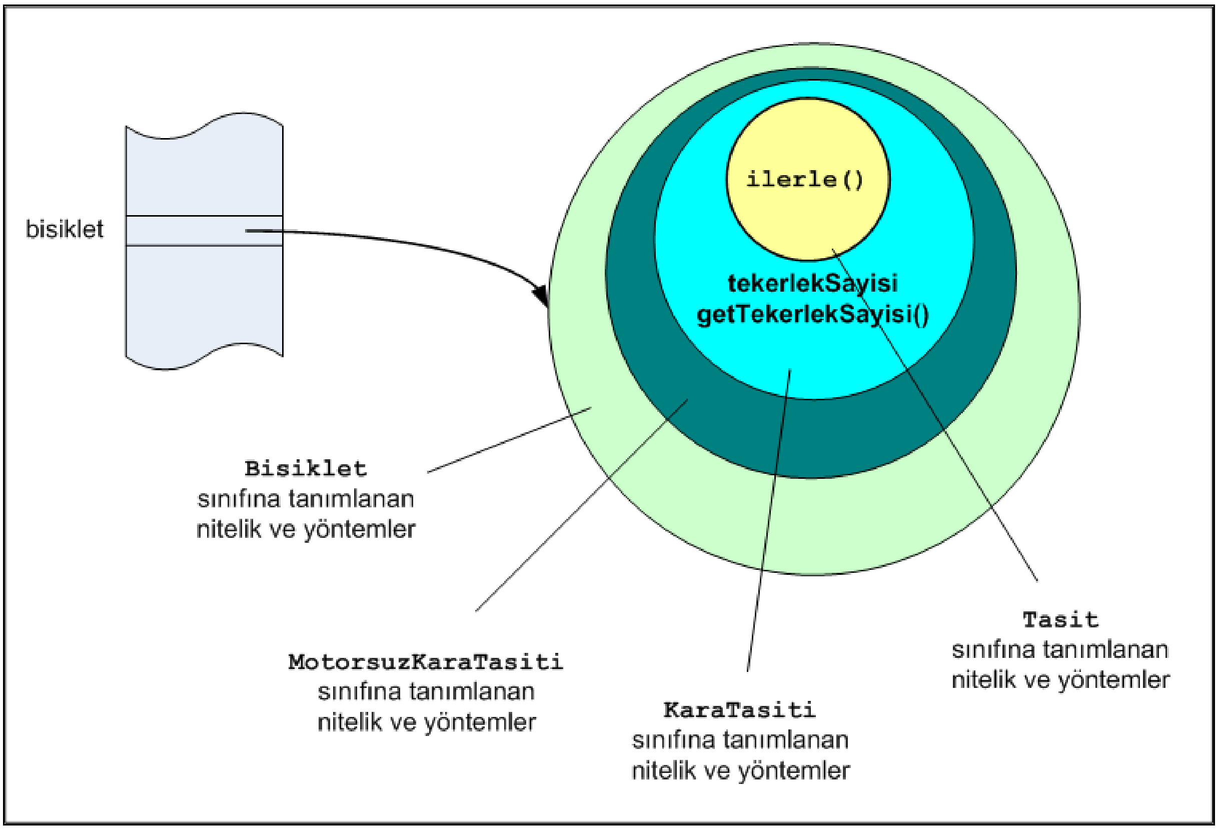
Bu program çalıştırıldığında, önce bir **Bisiklet** nesnesi oluşturulur. Sonra bu nesneye **ilerle(5)** iletisi gönderilir. **Bisiklet** sınıfının içinde (bkz. Kod 5-48) **ilerle()** yöntemi tanımlanmamış olmamasına rağmen ekranda "Taşıt 5 birim ilerliyor.." çıktısı oluşur. Çünkü **Bisiklet** sınıfı dolaylı olarak **Tasit** sınıfının alt sınıfıdır ve **Tasit** sınıfında tanımlanmış olan **ilerle()** yöntemi kalıtım ile **Bisiklet** sınıfına aktarılmıştır. Başka bir deyişle, **Bisiklet** sınıfının içinde **ilerle()** yöntemi kodlanmamış olsa bile, ata sınıfından aldığı **ilerle()** yöntemi zaten vardır ve bu yöntem işletilir.

Benzer şekilde, **bisiklet.getTekerlekSayisi()** çağrısı, **Bisiklet** sınıfının dolaylı olarak atası olan **KaraTasiti** sınıfından aldığı **getTekerlekSayisi()** yönteminin işletilmesine neden olur. Bu yöntem işletildiği zaman, **Bisiklet** sınıfının **KaraTasiti** sınıfından kalıtımla almış olduğu **tekerlekSayisi** niteliğinin değerini döndürecektir. Kodun hiçbir kesiminde **tekerlekSayisi** niteliğine değer atanmadığı ve **int** türünün varsayılan değeri 0 (sıfır) olduğu için, ekranda "Bisikletin 0 tekerleği var." yazılacaktır. Böylece programın çıktısı şöyle olur:

```
Taşıt 5 birim ilerliyor..
Bisikletin 0 tekerleği var.
```

Çıktı 5-6. Kalıtım örneği – Örnek programın çıktısı

Bunu Şekil 5-9 ile açıklamaya çalışalım: Görüldüğü gibi bellekteki bisiklet adlı referans değişken **`Bisiklet`** sınıfına ait olan ve yığında oluşturulan nesneyi göstermektedir. **`Bisiklet`** sınıfının atalarından aldığı niteilk ve yöntemler, nesnenin içinde farklı renkteki daireler içinde gösterilmiştir. **`Bisiklet`** tipindeki nesne, **`Tasit`** sınıfından gelen **`ileti()`** yöntemi ile **`KaraTasiti`** sınıfından gelen **`tekerlekSayisi`** niteliği ve **`getTekerlekSayisi()`** yöntemine sahiptir.



Şekil 5-9. Bisiklet sınıfının içi

Kalıtımda, alt sınıfın nesnesi oluşturulduğunda bu nesnenin içinde ata sınıfa bir iç-nesne de oluşturulur. İç-nesne de kendi ata sınıfına ait bir iç-nesneyi içerecek ve bu en üstteki sınıfa kadar zincirleme bir şekilde gidecektir. Şekilde 5-9'daki farklı renkli daireler aslında bu durumu göstermektedir.

Bir nesnenin içinde başka bir nesnenin var olması, daha önce anlattığımız "içerme (*composition*)" ilişkisi ile de olanaklıdır. Ancak kalıtımda farklı olan nokta, içerilen ata sınıfa ait nesnenin kendi arayüzünü (public ya da

protected tanımlanmış yöntem ve niteliklerini) alt sınıfa aktarmasıdır. Böylece alt sınıf, kendi atalarının da arayüzlerine sahip olur.

Sarmalama kodun değiştirilebilirliğini ve bakım kolaylığını, kalıtım ise kodun yeniden kullanılabilirliğini arttırır.

5.5.3 Kalıtımla Gelen Nitelik ve Yöntemlere Erişim

Kalıtım konusunda genelde en çok kafa karıştıran konu, ata sınıftan gelen nitelik ya da yöntemlere doğrudan erişimin bazen olanaklı olmamasıdır. Bir sınıf, kurucular hariç atasındaki herşeyi kalıtımla alır dedik. Bu ifadeye göre, **Bisiklet** sınıfı **KaraTasiti** sınıfından **tekerlekSayisi** niteliğini almaktadır. Bununla birlikte, **Bisiklet** sınıfı içinde **tekerlekSayisi** niteliğine doğrudan erişilemez. Çünkü bu nitelik, **KaraTasiti** sınıfında **private** tanımlanmıştır. **private** erişim düzenleyicisi, tanımlandığı sınıfın dışından erişimi engelleyen bir anahtar sözcüktür. Dolayısıyla, **KaraTasiti** sınıfı içinde tanımlanmış olan **tekerlekSayisi** niteliği, alt sınıfı oldukları halde **MotorluKaraTasiti** ya da **Bisiklet** sınıflarından doğrudan erişilemez.

Bir niteliğe doğrudan erişilemiyor olması, o niteliğin var olmadığı anlamına gelmez. **Bisiklet** sınıfının dolaylı olarak atası olan **KaraTasiti** sınıfından aldığı **tekerlekSayisi** niteliği vardır, ancak **Bisiklet** sınıfı bu niteliğe doğrudan erişemez. "Doğrudan erişmek" ifadesiyle, **Bisiklet** sınıfına ait kodun içerisinde **tekerlekSayisi** niteliğinin kodlanamayacağını belirtiyoruz. Aksi taktirde derleme hatası alınacaktır.

Aslında iç-nesneler düşünüldüğünde durum herhangi bir belirsizliğe ya da zorluğa yer bırakmayacak kadar açıklık kazanır. Bu iç-nesneler farklı sınıflara ait olduklarına göre ve farklı sınıflar arasındaki erişim kuralları da erişim düzenleyicilere göre belirlendiğine göre, hangi nitelik ya da yönteme nereden erişilebileceği rahatça görülebilir.

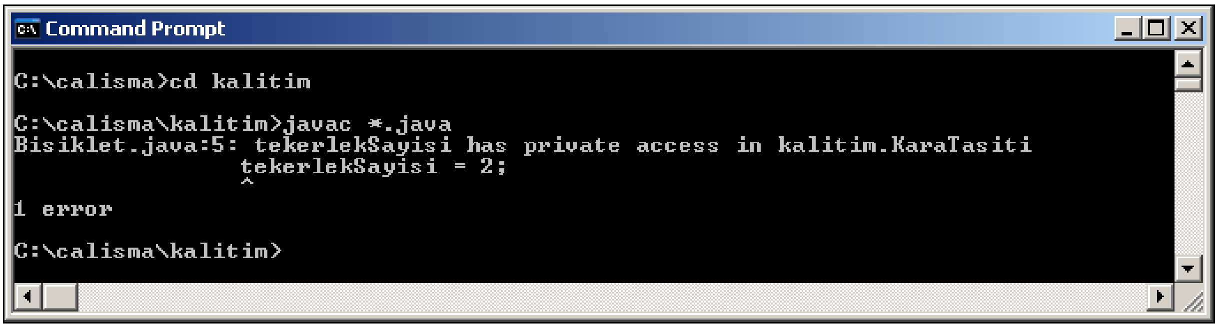
Aşağıdaki kod kesimiyle bunu örnekleyelim:

```
package kalitim;
```

```
public class Bisiklet extends MotorsuzKaraTasiti {
      public void tekerlekSayisiNiteligineErisim() {
            tekerlekSayisi = 2; // ! Derleme hatası
      }
}
```

Kod 5-52. Ata sınıfın private niteliğine erişim

Kodu derleyelim:



Şekil 5-10. Ata sınıfın private niteliğine erişim – derleme sonucu

Görüldüğü gibi **tekerlekSayisi** niteliğine **Bisiklet** sınıfı içinden erişmeye çalıştığımızda, derleyici bu niteliğin ata sınıfta **private** tanımlandığını belirten bir hata iletisi döndürüyor.

Bu durum Şekil 5-9'da da görülebilmektedir. **Bisiklet** sınıfının içinde iç-nesneleri belirtmek üzere farklı renklerde daireler kullanılmıştır. **private** nitelik ya da yöntemlere yalnızca kendi rengindeki dairelerin içinden erişilebildiği açıktır. **Bisiklet** sınıfının içinde atasından gelen **tekerlekSayisi** niteliği farklı renkteki bir dairenin içindedir. Demek ki bu niteliğe **Bisiklet** sınıfının içinden doğrudan erişilemez.

> ***Sınıflar arasında kalıtım ilişkisi olsa da erişim düzenleyiciler için tanımlanmış kurallar geçerlidir: private nitelik ya da yöntemlere tanımlandıkları sınıf dışından erişilemez!***

Dolayısıyla, alt sınıf içinde ata sınıftan gelen **private** niteliklere erişmek için, o niteliğin değerine erişimi denetimli olarak gerçekleştirmek üzere sarmalama ilkesi çerçevesinde kodlanmış yöntemleri kullanmamız gerekir.

İki sınıf arasında kalıtım iliişkisi kurulmuş olsa da sarmalama ilkesi için söylediğimiz herşey halen geçerlidir.

5.5.4 `protected` Erişim Düzenleyici

Erişim düzenleyicileri anlatırken, **`protected`** anahtar sözcüğünün paket erişimi için kullanıldığını söylemiştik. Ayrıca kalıtımda da farklı bir anlamı olduğunu ve bunu kalıtım konusu ile birlikte açıklayacağımızı belirtmiştik.

`protected` anahtar sözcüğü paket erişimini belirtmenin yanısıra, aralarında kalıtım ilişkisi bulunan sınıflar için farklı bir görev daha üstlenir: **`protected`** tanımlanan nitelik ya da yöntemlere doğrudan ya da dolaylı alt sınıflardan erişilebilir. Başka bir ifade ile **`protected`** alt sınıflar için **`public`**, diğer sınıflar için **`private`** gibi yorumlanabilir.

Örneğimizdeki **`KaraTasiti`** sınıfının **`tekerlekSayisi`** niteilğini **`protected`** yapalım. Öncelikle, örneğimizdeki diğer bütün sınıfların **`KaraTasiti`** sınıfı ile aynı pakette oldukları için **`tekerlekSayisi`** niteliğine doğrudan erişebileceklerini söyleyebiliriz:

```
package kalitim;
public class KaraTasiti extends Tasit {
      protected int tekerlekSayisi;
      public int getTekerlekSayisi() {
            return this.tekerlekSayisi;
      }
}
```

Kod 5-53. KaraTasiti sınıfında protected nitelik tanımlama

```
package kalitim;
public class DenizTasiti extends Tasit {
      public void protectedErisimOrnegi() {
            KaraTasiti k = new KaraTasiti();
            k.tekerlekSayisi = 5; // protected niteliğe erişim!!
      }
}
```

Kod 5-54. KaraTasiti sınıfı ile aynı paketteki bir sınıftan protected niteliğe erişim

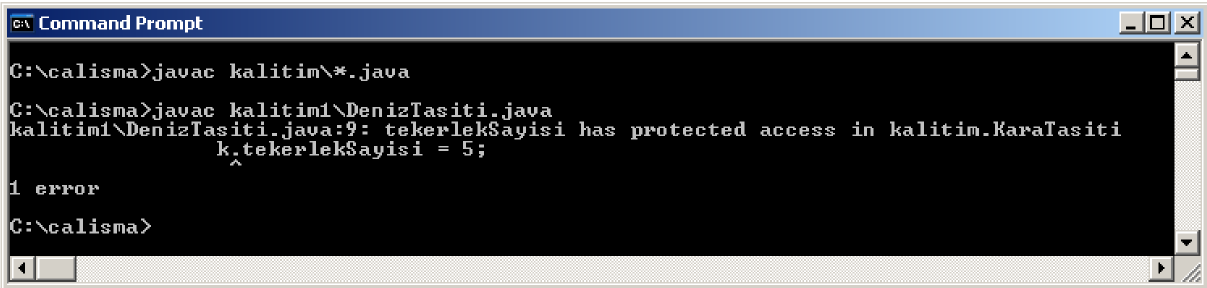
DenizTasiti sınıfı ile **KaraTasiti** sınıfı arasında kalıtım ilişkisi olmamasına rağmen (her ikisi de **Tasit** sınıfının alt sınıfıdır ama kendi aralarında bir ilişki tanımlanmamıştır), bu iki sınıf da aynı pakette olduğu için **DenizTasiti** sınıfındaki bir yöntemden **KaraTasiti** sınıfına ait nesnenin **tekerlekSayisi** niteliğine doğrudan erişilebilmektedir.

DenizTasiti sınıfını başka bir pakete taşıyalım:

```
package kalitim1;
import kalitim.KaraTasiti;
import kalitim.Tasit;
public class DenizTasiti extends Tasit {
      public void protectedErisimOrnegi() {
            KaraTasiti k = new KaraTasiti();
            k.tekerlekSayisi = 5;
      }
}
```

Kod 5-55. DenizTasiti sınıfının kalitim1 adlı bir pakete taşınması

Kodları yeniden derleyelim:



Şekil 5-11. Kodların yeniden derlenmesi

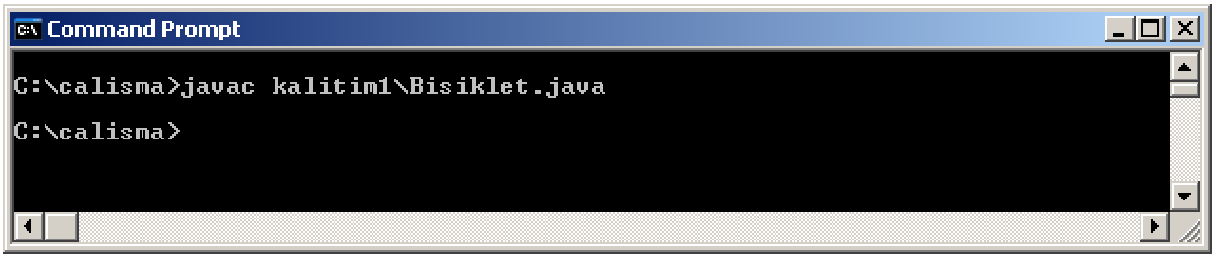
Görüldüğü gibi **DenizTasiti** sınıfı artık **kalitim1** adlı farklı bir pakette olduğu için **KaraTasiti** sınıfına ait **protected** tanımlı **tekerlekSayisi** niteliğine erişemez. Derleyici bu durumu belirten bir hata iletisi döndürmektedir.

Şimdi farklı pakette olsa da alt sınıftan **protected** tanımlanmış niteliğe erişilebileceğini gösterebilmek için **Bisiklet** sınıfını **kalitim1** paketine taşıyalım (bu arada **DenizTasiti** sınıfını da tekrar kalitim paketine taşıyarak derleme hatasından kurtulalım):

```
package kalitim1;
import kalitim.MotorsuzKaraTasiti;
public class Bisiklet extends MotorsuzKaraTasiti {
      public void tekerlekSayisiNiteligineErisim() {
            tekerlekSayisi = 2;
      }
}
```

Kod 5-56. Bisiklet sınıfının kalitim1 paketine taşınması

Kodu derleyelim:



Şekil 5-12. Kodların yeniden derlenmesi

`Bisiklet` sınıfı farklı bir pakette olduğu halde, dolaylı olarak atası olan **`KaraTasiti`** sınıfından gelen **`tekerlekSayisi`** adli niteliğe doğrudan erişebildi ve derlemede herhangi bir hata oluşmadı. Çünkü **`tekerlekSayisi`** niteliği **`KaraTasiti`** sınıfında **`protected`** tanımlanmış durumda ve **`protected`** tanımlı nitelik ya da yöntemlere, doğrudan ya da dolaylı alt sınıflardan erişilebilmektedir.

> ***`protected` erişim düzenleyicisi kalıtım ilişkisi söz konusu olduğunda `private` ve `public` erişimin arasında bir erişim sağlamaktadır. Ancak bu özelliğin kullanılabileceği durumlar pek sık oluşmamaktadır. `protected` erişimin aynı zamanda paket erişimine de izin verdiği unutulmamalı, dolayısıyla bu anahtar sözcük sarmalama ilkesine zarar vermeyecek şekilde dikkatle kullanılmalıdır.***

5.5.5 Kurucu Zinciri ve `super` Anahtar Sözcüğü

Bir sınıfa ait nesne oluşturulurken, o sınıfın bir kurucusunun işletildiğini, kurucunun çalışması tamamlandıktan sonra bellekte artık bir nesnenin oluştuğunu biliyoruz. Kurucuları da nesneleri ilk oluşturuldukları anda anlamlı durumlara taşıyabilmek için kullanıyoruz. Bu durumda, eğer

nesnesi oluşturulacak sınıf başka bir sınıfın alt sınıfıysa, önce ataya ait iç-nesnesinin oluşturulması ve bu nesnenin niteliklerinin ilk değerlerinin verilmesi gerektiğini söyleyebiliriz. Şekil 5-9'da görülen içiçe nesnelerin oluşabilmesi için nesnelerin içten dışa doğru oluşması gerekir.

İç-nesnenin oluşabilmesi için, nesnesi oluşturulacak sınıfa ait kurucu işletilmeye başladığı zaman ilk iş olarak ata sınıfa ait kurucu çağrılır. Eğer ata sınıf da başka bir sınıfın alt sınıfıysa, bu kez o sınıfın kurucusu çağrılır. Kurucu zinciri alt sınıftan ata sınıfa doğru bu şekilde ilerler. En üstte, kalıtım ağacının tepesindeki sınıfın kurucusunun çalışması sonlandıktan sonra sırası ile alt sınıfların kurucularının çalışması sonlanacaktır. Böylece içiçe nesneler sıra ile oluşturularak en son en dıştaki nesne oluşturulmuş olur ve kurucu zinciri tamamlanır.

Şimdi, **`Tasit`** örneğimize geri dönelim ve **`Bisiklet`** sınıfından başlayarak bütün atalarına parametre almayan birer kurucu yazalım:

```
package kalitim;
public class Bisiklet extends MotorsuzKaraTasiti {
      public Bisiklet() {
            System.out.println("Bisiklet sınıfının kurucusu");
      }
}
```

Kod 5-57. Kurucu zinciri – Bisiklet sınıfı

```
package kalitim;
public class MotorsuzKaraTasiti extends KaraTasiti {
      public MotorsuzKaraTasiti() {
            System.out.println("MotorsuzKaraTasiti sınıfının kurucusu");
      }
}
```

Kod 5-58. Kurucu zinciri – MotorsuzKaraTasiti sınıfı

```
package kalitim;
public class KaraTasiti extends Tasit {
      protected int tekerlekSayisi;

      public KaraTasiti() {
            System.out.println("KaraTasiti sınıfının kurucusu");
```

```
      }

      public int getTekerlekSayisi() {
            return this.tekerlekSayisi;
      }
}
```

Kod 5-59. Kurucu zinciri – KaraTasiti sınıfı

```
package kalitim;
public class Tasit {
      public Tasit() {
            System.out.println("Tasit sınıfının kurucusu");
      }
      public void ilerle(int birim) {
            System.out.println("Taşıt " + birim + " birim ilerliyor..");
      }
}
```

Kod 5-60. Kurucu zinciri – Tasit sınıfı

Şimdi de küçük bir programla sadece bir Bisiklet nesnesi oluşturalım:

```
package kalitim;
public class Program {
      public static void main(String[] args) {
            Bisiklet bisiklet = new Bisiklet();
      }
}
```

Kod 5-61. Kurucu zinciri – Program sınıfı

Bu program çalıştırıldığı zaman önce **`Bisiklet`** sınıfının kurucusu çağrılacaktır. Yukarıda anlattıklarımıza göre, **`Bisiklet`** sınıfının kurucusu ata sınıf olan **`MotorsuzKaraTasiti`** sınıfının kurucusunu çağırır. **`MotorsuzKaraTasiti`** sınıfının kurucusu, kendi atası olan **`KaraTasiti`** sınıfının kurucusunu, **`KaraTasiti`** sınıfının kurucusu ise **`Tasit`** sınıfının kurucusunu çağıracaktır. **`Tasit`** sınıfı başka bir sınıfın alt sınıfı olarak tanımlanmadığına göre **`Object`** sınıfıının alt sınıfıdır Eğer bir sınıf için **`extends`** ile kalıtım ilişkisi tanımlanmazsa o sınıf **`Object`** sınıfının alt sınıfı olur. Dolayısıyla bütün sınıflar **`Object`** sınıfının doğrudan ya da dolaylı alt sınıfıdır.) ve **`Object`** sınıfının kurucusunu çağırır. **`Object`** sınıfının kurucusu

işletildikten sonra **`Tasit`** sınıfının kurucusu işletilecek ve ekranda "`Tasit sınıfının kurucusu`" yazacaktır. Kurucu sonlanınca bunu çağıran kesime dönülecek ve ekrana "`KaraTasiti sınıfının kurucusu`" yazılacak, buradan dönülünce "`MotorsuzKaraTasiti sınıfının kurucusu`" yazılacak ve son olarak da "`Bisiklet sınıfının kurucusu`" yazılacaktır.

Çıktıya bakalım:

```
Tasit sınıfının kurucusu
KaraTasiti sınıfının kurucusu
MotorsuzKaraTasiti sınıfının kurucusu
Bisiklet sınıfının kurucusu
```

Çıktı 5-7. Kurucu zinciri – Program çıktısı

Bir sınıfa ait kurucu işletilirken, eğer kurucunun içine ata sınıfın hangi kurucusunun çağrılacağını belirten kod yazılmamışsa, ata sınıfın varsayılan kurucusu çağrılır (varsayılan kurucunun parametre almayan kurucu olduğunu ve eğer bir sınıfın içine hiç kurucu yazılmazsa derleyici tarafından bir varsayılan kurucu oluşturulduğunu hatırlayın). Bu durum, kodun içinde gizlice (*implicitly*) var olan **`super()`** çağrısı ile gerçekleştirilir. Yani eğer programcı kurucunun içine ata sınıfın hangi kurucusunun çağrılacağını kodlamazsa, derleyici tarafından ata sınıfın varsayılan kurucusunu çağıracak **`super();`** deyimi eklenir. Programcı tarafından kurucunun ilk deyimi olarak açıktan (*explicitly*) **`super()`** yazılması da varsayılan kurucunun çağrılmasını sağlayacaktır.

Örneğimize geri dönerek **`Bisiklet`** sınıfının kurucusunda ata sınıfın varsayılan kurucusunu açıktan çağıralım:

```
package kalitim;
public class Bisiklet extends MotorsuzKaraTasiti {
      public Bisiklet() {
            super();    // ata sınıfın varsayılan kurucusu çağrılıyor
            System.out.println("Bisiklet sınıfının kurucusu");
      }
}
```

Kod 5-62. super anahtar sözcüğünün açıktan kullanılışı

Eğer ata sınıfta varsayılan kurucu yoksa ve programcı alt sınıftaki kurucunun içinde ata sınıfın hangi kurucusunun çağrılacağını belirtmezse derleme hatası alınacaktır. Çünkü derleyici aksi belirtilmedikçe ata sınıfın varsayılan kurucusunu çağıran **`super()`** kodunu üretecektir.

Ata sınıfın hangi kurucusunun çağrılacağı, **`super`** anahtar sözcüğü ile birlikte verilen parametrelere göre belirlenir. Nasıl ki **`new`** işleci ile birlikte kullandığımız parametreler hangi kurucunun çağrılacağını belirliyorsa, **`super`** anahtar sözcüğü ile birlikte kullanılan parametreler de aynı şekilde ata sınıfın hangi kurucusunun işletileceğini belirler.

Örneğin, **`KaraTasiti`** sınıfımızda tekerlek sayısını parametre olarak alan bir kurucu yazalım ve diğer kurucuyu silelim:

```
package kalitim;
public class KaraTasiti extends Tasit {
      protected int tekerlekSayisi;

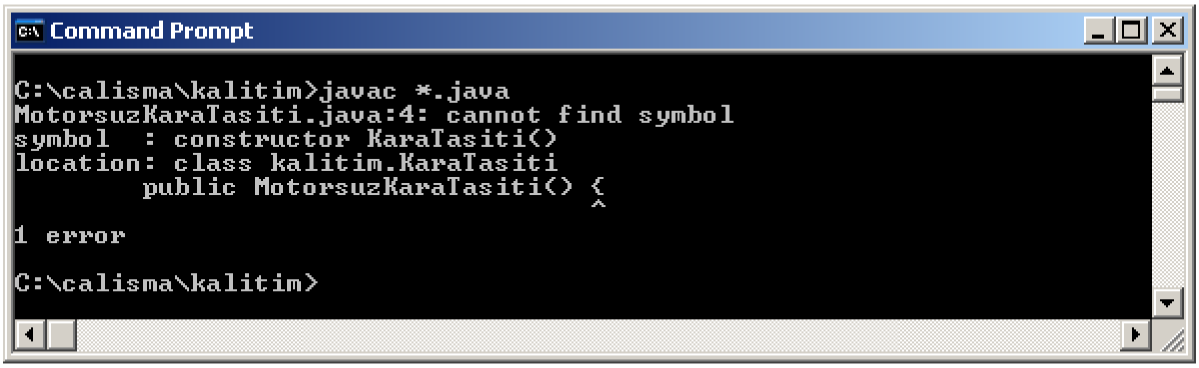
      public KaraTasiti(int tekerlekSayisi) {
            this.tekerlekSayisi = tekerlekSayisi;
            System.out.println("KaraTasiti sınıfının kurucusu");
      }

      public int getTekerlekSayisi() {
            return this.tekerlekSayisi;
      }
}
```

Kod 5-63. super anahtar sözcüğü - KaraTasiti sınıfı

Artık **`KaraTasiti`** sınıfına bir kurucu yazdığımıza göre derleyici tarafından varsayılan kurucu üretilmeyecektir.

Şimdi diğer sınıfların kodlarını değiştirmeden yeniden derleyelim:



Şekil 5-13. Kodların yeniden derlenmesi

KaraTasiti sınıfında artık parametresiz bir kurucu olmadığı ve **MotorsuzKaraTasiti** sınıfının kurucusunda ata sınıfın parametreli kurucusunu çağıracak herhangi bir kod yazılmadığından derleme hatası aldık.

Şimdi **MotorsuzKaraTasiti** ve **Bisiklet** sınıflarına, tekerlek sayısını parametre alan kurucuları ekleyelim:

```
package kalitim;
public class MotorsuzKaraTasiti extends KaraTasiti {
      public MotorsuzKaraTasiti(int tekerlekSayisi) {
            super(tekerlekSayisi);
            System.out.println("MotorsuzKaraTasiti sınıfının kurucusu");
      }


}
```

Kod 5-64. super anahtar sözcüğü – MotorsuzKaraTasiti sınıfı

```
package kalitim;
public class Bisiklet extends MotorsuzKaraTasiti {
      public Bisiklet(int tekerlekSayisi) {
            super(tekerlekSayisi);
            System.out.println("Bisiklet sınıfının kurucusu");
      }
}
```

Kod 5-65. super anahtar sözcüğü - Bisiklet sınıfı

Programımızı da **bisiklet** nesnesini oluştururken tekerlek sayısını belirleyecek şekilde değiştirelim:

```
package kalitim;
```

```
public class Program {
      public static void main(String[] args) {
            Bisiklet bisiklet = new Bisiklet(2);
            System.out.println("Bisikletin " +
                  bisiklet.getTekerlekSayisi() + " tekerleği vardır..");
      }
}
```

Kod 5-66. super anahtar sözcüğü – Program sınıfı

Programı işlettiğimizde oluşan çıktı:

```
Tasit sınıfının kurucusu
KaraTasiti sınıfının tekerlek sayısını parametre alan kurucusu
MotorsuzKaraTasiti sınıfının kurucusu
Bisiklet sınıfının kurucusu
Bisikletin 2 tekerleği vardır..
```

Çıktı 5-8. super anahtar sözcüğü – Program çıktısı

`Program` çalıştırıldığı zaman kurucu zinciri gene aynı sırada işletildi. Ancak bu sefer **`super`** anahtar sözcüğünü açıktan kullanarak hangi kurucunun çalıştırılacağını belirlemek zorunda kaldık. Programımıza eklediğimiz yeni satırla da **`bisiklet`** nesnemizin atalarından birisi olan **`KaraTasiti`** sınıfının kurucusunda değeri belirlenen **`tekerlekSayisi`** niteliğinin değerini ekrana yazdırdık. Bu değer **`Bisiklet`** sınıfının kurucusuna parametre olarak verilmişti. Ancak **`super(tekerlekSayisi)`** deyimi ile bu değeri önce **`MotorsuzKaraTasiti`** sınıfının kurucusuna, oradan da **`KaraTasiti`** sınıfının kurucusuna göndermiş olduk.

5.5.6 Yöntemlerin Geçersiz Kılınması (*Method Overriding*)

Aynı isimde farklı parametre listesi (sayı, tür ya da sıraları farklı olan parametreler) ile birden fazla yöntem kodlanmasını Yöntemlerin Aşırı Yüklenmesi (*Method Overloading*) olarak adlandırmıştık.

Aşırı yükleme, bir sınıfın içine ata sınıflarındaki **`protected`** ya da **`public`** tanımlanmış yöntemlerle aynı isimde ve farklı parametre listesine sahip yeni yöntemler yazılarak da gerçekleştirilebilir. Çünkü bu sınıf ata sınıfındaki yöntemleri kalıtımla alacaktır.

Yöntem Geçersiz Kılma ise bir alt sınıfın içine doğrudan ya da dolaylı ata sınıflarından gelen bir (ya da daha fazla) yöntemin aynısının (aynı yöntem adı ve aynı parametre listesi) kodlanmasına verilen addır.

Yöntem geçersiz kılma ile ilgili genelde şuna benzer sorular sorulur: "ata sınıftan zaten alınan yöntemin aynısını alt sınıfta neden tekrar kodlarım?", "kalıtım kodun yeniden kullanılabilirliğini arttırırken, benim alt sınıfa aynı yöntem(ler)i yeniden kodlamam çelişkili değil mi?".

Yöntem geçersiz kılma ile ilgili olarak bir noktanın gözden kaçırılmaması gerekir: **alt sınıfa kodlanan yöntem, ata sınıftaki yöntemle aynı ad ve parametre listesine sahiptir, ancak ata sınıftaki yöntemle aynı kodları içermemelidir!** Zaten alt sınıfa ata sınıftaki yöntemin tamamen aynısını kodlamak elbette çelişkili, hatta saçma ve anlamsız olacaktır.

Geçersiz kılmanın neden gerekli olduğunu anlayabilmek için öncelikle kalıtım ağacında aşağıya doğru inildikçe daha özel sınıflara, yukarıya doğru çıkıldıkça daha genel sınıflara ulaşıldığını hatırlamamız gerekir. Ata sınıfta tanımlanan bir yöntem, o sınıfın genelleştirdiği bütün alt sınıfların ortak özelliklerine göre çalışan bir yöntem olacaktır. Alt sınıflara inildikçe sınıflar özelleştiği için, ata sınıftaki yöntem alt sınıf için fazla genel ve dolayısıyla yetersiz kalabilir. Bu durumda alt sınıf, kendi özelliklerine bağlı olarak daha özel bir gerçekleştirim yapacaktır.

Bazen bu gerçekleştirim ata sınıftakini kullanıp üzerine birşeyler ekleyecek, bazen de tamamen farklı olacak şekilde kodlanabilir. Eğer alt sınıftaki gerçekleştirim ata sınıftaki yöntemi kullanacak ve üzerine birşeyler ekleyecekse, **`super`** anahtar sözcüğü atadaki yöntemi çağırmak üzere kullanılabilir. **`this`** anahtar sözcüğünün içinde bulunulan nesneye referans olması gibi, **`super`** anahtar sözcüğü de ata sınıfa ait iç nesneye referanstır.

Şimdi **`Tasit`** örneğimize geri dönelim ve sınıflarımızın içine ekrana o sınıftan oluşturulan nesnelerle ilgili bilgi döndürmek üzere **`bilgiVer()`** yöntemlerini kodlayalım:

```
package kalitim;
```

```java
public class Tasit {
	public Tasit() {
	}
	public void ilerle(int birim) {
		System.out.println("Taşıt " + birim + " birim ilerliyor..");
	}
	public String bilgiVer() {
		return "Tasit";
	}
}
```

Kod 5-67. Yöntem Geçersiz Kılma – Tasit sınıfı

```java
package kalitim;
public class KaraTasiti extends Tasit {
	private int tekerlekSayisi;

	public KaraTasiti(int tekerlekSayisi) {
		this.tekerlekSayisi = tekerlekSayisi;
	}
	public int getTekerlekSayisi() {
		return this.tekerlekSayisi;
	}
	public String bilgiVer() {
		return super.bilgiVer() + " - KaraTasiti";
	}
}
```

Kod 5-68. Yöntem Geçersiz Kılma – KaraTasiti sınıfı

```java
package kalitim;
public class MotorsuzKaraTasiti extends KaraTasiti {
	public MotorsuzKaraTasiti(int tekerlekSayisi) {
		super(tekerlekSayisi);
	}
	public String bilgiVer() {
		return super.bilgiVer() + " - MotorsuzKaraTasiti";
	}
}
```

Kod 5-69. Yöntem Geçersiz Kılma – MotorsuzKaraTasiti sınıfı

```java
package kalitim;
public class Bisiklet extends MotorsuzKaraTasiti {
```

```
      public Bisiklet(int tekerlekSayisi) {
            super(tekerlekSayisi);
      }
      public String bilgiVer() {
            return super.bilgiVer() + " - Bisiklet";
      }
}
```

Kod 5-70. Yöntem Geçersiz Kılma – Bisiklet sınıfı

Şimdi **`Bisiklet`** sınıfı ile **`Tasit`** sınıfından birer nesne oluşturup, bu nesnelere ait bilgileri ekrana yazmak üzere **`bilgiVer()`** yöntemlerini çağıran programı kodlayalım.

```
package kalitim;
public class Program {
      public static void main(String[] args) {
            Bisiklet bisiklet = new Bisiklet(2);
            System.out.println("Bisiklet'in bilgileri: " +
                bisiklet.bilgiVer());

            Tasit tasit = new Tasit();
            System.out.println("Tasit'in bilgileri: " + tasit.bilgiVer());
      }
}
```

Kod 5-71. Yöntem Geçersiz Kılma – Program sınıfı

Bu program işletildiği zaman, **`Bisiklet`** sınıfından oluşturulan nesne üzerinden çağrılan **`bilgiVer()`** yöntemi, önce doğrudan atası olan **`MotorsuzKaraTasiti`** sınıfının **`bilgiVer()`** yöntemini çağırır. **`MotorsuzKaraTasiti`** sınıfındaki **`bilgiVer()`** yöntemi benzer şekilde **`KaraTasiti`** sınıfındaki **`bilgiVer()`** yöntemini, o da **`Tasit`** sınıfındaki **`bilgiVer()`** yöntemini çağırır. **`Tasit`** sınıfının **`bilgiVer()`** yönteminden geriye `"Tasit"` döner, **`KaraTasiti`** sınıfı buna " – KaraTasiti" ekleyip döndürür, **`MotorsuzKaraTasiti`** sınıfı buna `" – MotorsuzKaraTasiti"` ekleyip döndürür ve en son **`Bisiklet`** sınıfında buna `" – Bisiklet"` eklenip döndürülür. Dönen sonuç **`Program`** sınıfında ekrana yazdırılır: `"Bisiklet'in bilgileri: Tasit - KaraTasiti - MotorsuzKaraTasiti – Bisiklet"`. **`Tasit`**

sınıfından oluşturulan nesneye gönderilen **bilgiVer()** iletisinden ise "Tasit" döner ve **Program** sınıfında ekrana "Tasit'in bilgileri: Tasit" yazdırılır. Programın çıktısı:

```
Bisiklet'in bilgileri: Tasit - KaraTasiti - MotorsuzKaraTasiti - Bisiklet
Tasit'in bilgileri: Tasit
```

Çıktı 5-9. Yöntem Geçersiz Kılma – Program çıktısı

Görüldüğü gibi **Tasit** sınıfından alt sınıflarına doğru inildikçe **bilgiVer()** yöntemi sınıfa bağlı olarak özelleşmektedir ve geçersiz kılma ile bu özelleşme ele alınmıştır. Örnekte ayrıca **super** anahtar sözcüğünün atadaki yönteme ulaşmak üzere nasıl kullanıldığı da görülmektedir.

5.5.7 Yöntemlerde Geçersiz Kılmanın `final` ile Engellenmesi

final anahtar sözcüğünün sabit tanımlamada nasıl kullanıldığını daha önce görmüştük. Eğer **final** sözcüğü bir nitelikten önce kullanılırsa, o niteliğe bir kez değer atandıktan sonra niteliğin değeri değiştirilemez.

final sözcüğünün bir başka kullanım alanı, geçersiz kılınması istenmeyen yöntemlerin belirlenmesidir. Bir sınıfta tanımlanan yöntemin alt sınıflarda geçersiz kılınması istenmiyorsa, o yöntem **final** tanımlanır. **final** tanımlanan yöntemlerin alt sınıflarda geçersiz kılınamayacağı derleyici tarafından garanti edilir.

5.5.8 `Object` sınıfı

Java'da bütün sınıflar **Object** sınıfının alt sınıfıdır.

Yazdığımız herhangi bir sınıfın tanımına **extends Object** yazmasak bile, bu sınıf için **extends Object** tanımlıdır. Dolayısıyla bütün sınıflar doğrudan ya da dolaylı olarak **Object** sınıfının alt sınıfıdır. Böylece bütün nesnelerin bu sınıftaki yöntemleri içermesi sağlanmış olur.

Şimdi bu sınıftaki bazı yöntemleri inceleyelim:

protected Object clone(): Nesnenin bir kopyasını oluşturur ve döndürür.

public boolean equals(Object obj): Parametre olarak verilen nesnenin bu yöntem üzerinden çağrıldığı nesneye eşit olup olmadığını bulur ve döndürür. Bildiğimiz gibi eşitlik denetimi yapan işleç "**==**" işlecidir. Ancak bu işleç nesne referansları için kullanılırsa nesnelerin eşitliğini değil, nesneleri gösteren referansların eşitliğini, yani referansıların aynı adresi gösterip göstermediklerini denetler. İki nesnenin birbirine eşit olması, bu nesnelerin referanslarının aynı adresi göstermesi demek değildir. İki nesnenin eşitliği, nesnelerin ait oldukları sınıflara göre denetlenmelidir. Örneğin iki **Zaman** nesnesinin eşit olması demek, **saat**, **dakika** ve **saniye** niteliklerinin değerlerinin eşit olması demektir. Dolayısıyla iki **Zaman** nesnesinin eşit olup olmadıklarını anlamak üzere **Zaman** sınıfının içinde **equals** yönteminin kodlanması (geçersiz kılınması) gerekir.

public String toString(): Nesnenin **String** gösterimini oluşturur ve döndürür. **Object** sınıfındaki gerçekleştirimi nesnenin ait olduğu sınıfı ve bellekteki adresini döndürür. **String** gösterimi istenen sınıflar için **toString** yönteminin geçersiz kılınması gerekir. **Tasit** örneğimizdeki **bilgiVer** yöntemlerinin adlarını **toString** yapalım:

```
package kalitim;
public class Tasit {
      public Tasit() {
      }
      public void ilerle(int birim) {
            System.out.println("Taşıt " + birim + " birim ilerliyor..");
      }
      public String toString() {
            return "Tasit";
      }
}
```

Kod 5-72. toString() yöteminin geçersiz kılınması – Tasit sınıfı

```
package kalitim;
public class KaraTasiti extends Tasit {
      private int tekerlekSayisi;

      public KaraTasiti(int tekerlekSayisi) {
```

```java
		this.tekerlekSayisi = tekerlekSayisi;
	}
	public int getTekerlekSayisi() {
		return this.tekerlekSayisi;
	}
	public String toString() {
		return super.toString() + " - KaraTasiti";
	}
}
```

Kod 5-73. toString() yöteminin geçersiz kılınması – KaraTasiti sınıfı

```java
package kalitim;
public class MotorsuzKaraTasiti extends KaraTasiti {
	public MotorsuzKaraTasiti(int tekerlekSayisi) {
		super(tekerlekSayisi);
	}
	public String toString() {
		return super.toString() + " - MotorsuzKaraTasiti";
	}
}
```

Kod 5-74. toString() yöteminin geçersiz kılınması – MotorsuzKaraTasiti sınıfı

```java
package kalitim;
public class Bisiklet extends MotorsuzKaraTasiti {
	public Bisiklet(int tekerlekSayisi) {
		super(tekerlekSayisi);
	}
	public String toString() {
		return super.toString() + " - Bisiklet";
	}
}
```

Kod 5-75. toString() yöteminin geçersiz kılınması – Bisiklet sınıfı

```java
package kalitim;
public class Program {
	public static void main(String[] args) {
		Bisiklet bisiklet = new Bisiklet(2);
		System.out.println("Bisiklet'in bilgileri: " + bisiklet);

		Tasit tasit = new Tasit();
		System.out.println("Tasit'in bilgileri: " + tasit);
```

```
        }
}
```

Kod 5-76. toString() yöteminin geçersiz kılınması – Program sınıfı

Program sınıfında, nesneleri ekrana yazdıran satırlara bakalım. Bu satırlarda **`toString()`** yöntemi açıktan çağrılmadığı halde, **`String`**'ten sonra kullanılan "+" işleci bitiştirme işlemi yaptığı için `"Bisiklet'in bilgileri:"` **`String`**'inden sonra gelen "+" işleci, **`bisiklet`** nesnesinin **`toString()`** yönteminin gizlice çağrılmasını sağlar. Aynı şekilde **`tasit`** nesnesinin de **`toString()`** yöntemi çağrılmaktadır. Programın çıktısı aşağıdaki gibi olacaktır:

```
Bisiklet'in bilgileri: Tasit - KaraTasiti - MotorsuzKaraTasiti - Bisiklet
Tasit'in bilgileri: Tasit
```

Çıktı 5-10. toString() yöteminin geçersiz kılınması – Program çıktısı

5.6 Çokbiçimlilik (*Polymorphism*)

Nesneye yönelik programlamanın 3. aracı çokbiçimliliktir.

Çokbiçimlilik, nesneye yönelik yaklaşımla geliştirilen uygulamaların genişletilebilirliğini (*extensibility*) sağlayan çok önemli bir araçtır.

Bu bölümde çokbiçimliliğin nasıl gerçekleştirildiği açıklanacaktır.

5.6.1 Ata Sınıf Referansından Alt Sınıf Nesnesine Ulaşma

Eğer **`KaraTasiti`** sınıfı **`Tasit`** sınıfının alt sınıfı olarak tanımlanmışsa, "Her **`KaraTasiti`** bir **`Tasit`**'tir" önermesi doğru olmalıdır. Öyleyse **`Tasit`** sınıfından tanımlanmış bir referans değişkenin **`KaraTasiti`** sınıfına ait bir nesneyi göstermesi de mantıksal olarak doğru olacaktır. Nesneye yönelik programlama dilleri de bu mantıksal yaklaşımı desteklemektedir:

```
package cokbicimlilik;
public class Calgi {
}
```

Kod 5-77. Ata sınıftan alt sınıf nesnesine erişim – Calgi sınıfı

```
package cokbicimlilik;
public class TelliCalgi extends Calgi {
```

```
}
```

Kod 5-78. Ata sınıftan alt sınıf nesnesine erişim – TelliCalgi sınıfı

```
package cokbicimlilik;
public class Gitar extends TelliCalgi {
}
```

Kod 5-79. Ata sınıftan alt sınıf nesnesine erişim – Gitar sınıfı

```
package cokbicimlilik;
public class Program {
      public static void main(String[] args) {
            Calgi calgi1 = new Gitar();
            TelliCalgi calgi2 = new Gitar();
            Calgi calgi3 = new TelliCalgi();
      }
}
```

Kod 5-80. Ata sınıftan alt sınıf nesnesine erişim – Program sınıfı

Bu örnekte 3 derinlikte bir kalıtım ağacı görülmektedir: **`Calgi`** sınıfının alt sınıfı olan **`TelliCalgi`** sınıfı ve onun da alt sınıfı olan **`Gitar`** sınıfı. **`Program`** sınıfında ise üç tane nesne oluşturulmaktadır. Program herhangi bir işlem gerçekleştirmemektedir. Amaç yalnızca, ata sınıftan tanımlanan referanslar ile alt sınıfa ait olarak oluşturulan nesnelerin tutulabildiğini göstermektir.

5.6.2 Geç Bağlama (*Late Binding*)

Yapısal programlama dillerinde, derleyici (*compiler*) tarafından herbir işlev için ayrı bir çağrı oluşturulur ve bağlayıcı (*linker*) bu çağrıları işletilecek kod için mutlak adreslere dönüştürür (Bkz. Şekil 4.1). Bu sayede, çalıştırılabilir dosya oluştuğunda bütün işlev çağrıları ile ilgili adresler belirlidir. Buna "erken bağlama (*early binding*)" adı verilir.

Nesneye yönelik programlarda ise yöntem çağrıları nesnelere gönderilen iletilerden oluşur. Nesneler çalışma zamanında oluşturuldukları için yöntem çağrılarının hangi adreslerde olacağı çalışma zamanına kadar bilinemez.

Bu nedenle nesneye yönelik programlama dilleri, üreysel (*generic*) bir tür üzerinden yöntem çağrılmasına izin verir. Derleyici, öyle bir yöntemin var olup olmadığı, parametrelerin ve dönüş değerlerinin tür denetimlerini gerçekleştirebilir ancak yöntem çağrısının hangi nesne üzerinden işletileceği çalışma zamanında belirlenir. Dolayısıyla hangi kodun işletileceği derleme zamanında bilinmemektedir. Buna "geç bağlama (*late binding*)" adı verilir.

Bazı programlama dillerinde geç bağlama yapılacağını programcının kodlayacağı anahtar sözcükler belirlerken (C++ dilindeki **`virtual`** anahtar sözcüğü gibi), Java'da bütün yöntem çağrıları için varsayılan yaklaşım geç bağlamanın kullanılmasıdır. Dolayısıyla Java'da geç bağlama için herhangi bir özel anahtar sözcüğe gereksinim yoktur.

5.6.3 Çokbiçimlilik Nasıl Gerçekleşir?

Şimdi bir kod örneği ile bu anlattıklarımızı birleştirelim.

```
package cokbicimlilik;
public class Calgi {
	public void cal() {
		System.out.println("Ses..");
	}
}
```

Kod 5-81. Çokbiçimlilik örneği – Calgi sınıfı

```
package cokbicimlilik;
public class TelliCalgi extends Calgi {
}
```

Kod 5-82. Çokbiçimlilik örneği – TelliCalgi sınıfı

```
package cokbicimlilik;
public class Gitar extends TelliCalgi {
	public void cal() {
		System.out.println("Gitar sesi..");
	}
}
```

Kod 5-83. Çokbiçimlilik örneği – Gitar sınıfı

```
package cokbicimlilik;
```

```
public class Program {
      public static void main(String[] args) {
            Calgi calgi = new Calgi();
            calgi.cal();

            calgi = new Gitar();
            calgi.cal();
      }
}
```

Kod 5-84. Çokbiçimlilik örneği – Program sınıfı

Görüldüğü gibi **`Calgi`** sınıfı, onun alt sınıfı olan **`TelliCalgi`** sınıfı ve onun alt sınıfı olan **`Gitar`** sınıfı tanımlanmış. **`Calgi`** sınıfında tanımlanan **`cal()`** yöntemi **`Gitar`** sınıfında geçersiz kılınmış.

`Program` sınıfında önce **`Calgi`** sınıfından bir referans değişken tanımlanmış ve bu referans değişkenin içine **`Calgi`** sınıfına ait bir nesnenin referansı koyulmuş. Daha sonra da bu referans değişken kullanılarak **`cal()`** yöntemi çağrılmış. Ekranda "`Ses..`" çıktısının oluşmasını bekliyoruz.

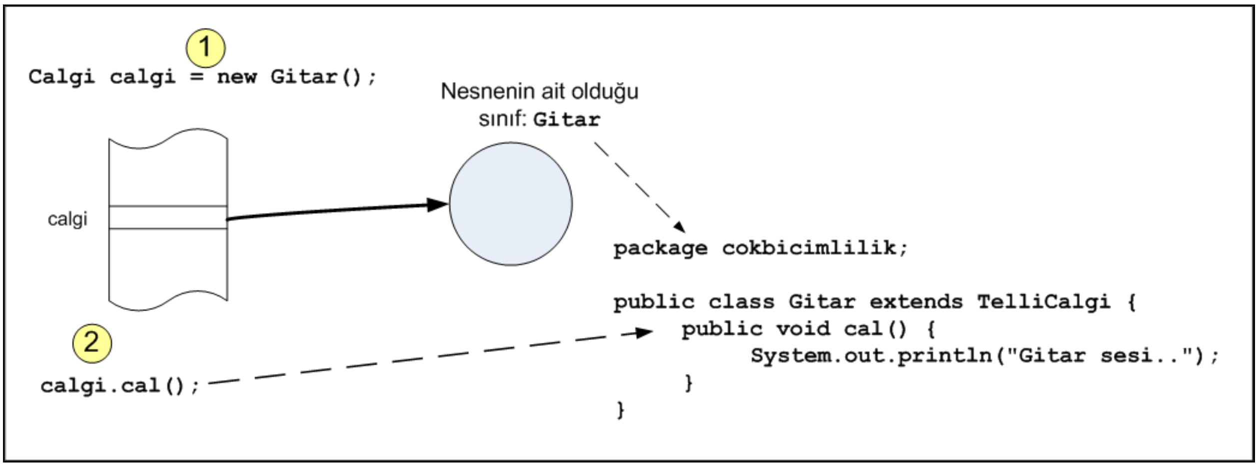
Sonraki satırda aynı referans değişkene bir **`Gitar`** nesnesinin referansı aktarılmış. Bunun olanaklı olduğunu, yani ata sınıf türünden tanımlanan bir referans değişkenin içinde alt sınıfa ait bir nesnenin referansının saklanabileceğini söylemiştik. Daha sonra bu referans üzerinden **`cal()`** yöntemi çağrılmış.

Burada koda iki farklı noktadan tekrar bakalım.

- *Derleyici açısından*: elimizdeki referans değişkenin türü **`Calgi`** sınıfıdır. Öyleyse bu değişken ile yalnızca **`Calgi`** sınıfında bulunan (**`Calgi`** sınıfının içine kodlanan ya da **`Calgi`** sınıfının kendi üst sınıflarından aldığı) yöntemler çağrılabilir. Dolayısıyla, **`calgi.cal()`** yönteminin derleme hatasına neden olmaması için, **`Calgi`** sınıfının içinde **`cal()`** yönteminin tanımlanmış olması gerekir. Gerçekten de **`Calgi`** sınıfından böyle bir yöntem tanımlanmış olduğuna göre bu yöntem çağrısında herhangi bir sorun yoktur.

- *Geç bağlama açısından*: referans değişkenin türü **`Calgi`** sınıfı olsa da gösterdiği yerde **`Gitar`** sınıfının nesnesi bulunmaktadır. Hangi yöntemin çağrılacağı çalışma zamanında belli olduğuna ve çalışma zamanında bir **`Gitar`** nesnesi bulunduğuna göre, **`calgi.cal()`** yöntem çağrısı, **`calgi`** referans değişkeninin gösterdiği yerde bulunan **`Gitar`** nesnesi üzerinden işletilecek demektir. **`Gitar`** nesnesi ise **`Gitar`** sınıfına aittir ve bu sınıfın içinde **`cal()`** yöntemi kodlanmıştır. Öyleyse bu satırın çıktısı `"Gitar sesi.."` şeklinde olacaktır.

Bu durumu aşağıdaki şekille açıklamaya çalışalım:



Şekil 5-14. Yöntemin çalışma zamanında oluşturulan nesne üzerinden işletilmesi

Programın çıktısı şu şekilde olacaktır:

```
Ses..
Gitar sesi..
```

Çıktı 5-11. Çokbiçimlilik örneği – Program çıktısı

Dikkat edilirse, **`calgi.cal();`** deyimi iki satırda bulunmaktadır ve bu iki satır işletildiğinde aynı deyim çalışmasına rağmen birbirinden farklı çıktı üretilmiştir. Yani aynı deyim, birden fazla biçimde çalışmaktadır. Dolayısıyla bu duruma çokbiçimlilik adı verilmektedir.

Soru:

Geri kalan kodlarda herhangi bir değişiklik yapmadan yalnızca **`Gitar`** sınıfında değişiklik yapılsa ve **`Gitar`** sınıfında **`cal()`** yöntemi geçersiz

kılınmasa ne olur? Yani **Gitar** sınıfımızın kodu aşağıdaki gibi olsa program derleme hatası verir mi? Vermezse programın çıktısı ne olur?

```
package cokbicimlilik;
public class Gitar extends TelliCalgi {
}
```

Kod 5-85. Çokbiçimlilik örneği – Gitar sınıfında cal() yöntemi geçersiz kılınmıyor

Yanıt:

Kodu derleyip işletmeden önce bildiklerimizi kullanarak ne olacağını söylemeye çalışalım:

```
      Calgi calgi = new Calgi();
      calgi.cal();
```

kesiminin gene aynı şekilde çalışacağını biliyoruz.

```
      calgi = new Gitar();
      calgi.cal();
```

kesiminde ise gene derleyici açısından herhangi bir sorun bulunmuyor. Geç bağlama kavramını düşündüğümüzde, **cal()** yönteminin gene **Gitar** sınıfından bir nesne üzerinden işletileceğini biliyoruz. Ancak bu sefer **Gitar** sınıfında **cal()** yöntemi geçersiz kılınmadığına göre, atadan gelen **cal()** yöntemi işletilecektir. **Gitar** sınıfının doğrudan atası olan **TelliCalgi** sınıfında da **cal()** yöntemi kodlanmamış olduğuna göre bu yöntem **Calgi** sınıfından gelen **cal()** yöntemi olacaktır. Öyleyse programın çıktısı şu şekilde olmalıdır:

```
Ses..
Ses..
```

Çokbiçimliliğin gerçekleşebilmesi için;

1- Kalıtım ilişkisi tanımlı olmalı

2- Ata sınıf türünden tanımlanmış referans değişken ile alt sınıf türünden oluşturulmuş nesneye ulaşılmalı

3- Alt sınıf, atasındaki yöntem(ler)i geçersiz kılmış olmalıdır.

Soru:

Şimdi **Gitar** sınıfımıza **cal()** yöntemini yeniden kodlayalım ve bu sefer **Calgi** sınıfından **cal()** yöntemini silelim. Program sınıfında da **main()** yönteminin ilk iki satırını silip kodu şu duruma getirelim:

```
package cokbicimlilik;
public class Calgi {
}
```

Kod 5-86. Soru – Calgi sınıfı

```
package cokbicimlilik;
public class TelliCalgi extends Calgi {
}
```

Kod 5-87. Soru – TelliCalgi sınıfı

```
package cokbicimlilik;
public class Gitar extends TelliCalgi {
      public void cal() {
            System.out.println("Gitar sesi..");
      }
}
```

Kod 5-88. Soru – Gitar sınıfı

```
package cokbicimlilik;
public class Program {
      public static void main(String[] args) {
            Calgi calgi = new Gitar();
            calgi.cal();
      }
}
```

Kod 5-89. Soru – Program sınıfı

Bu kodun ne yapmasını beklersiniz?

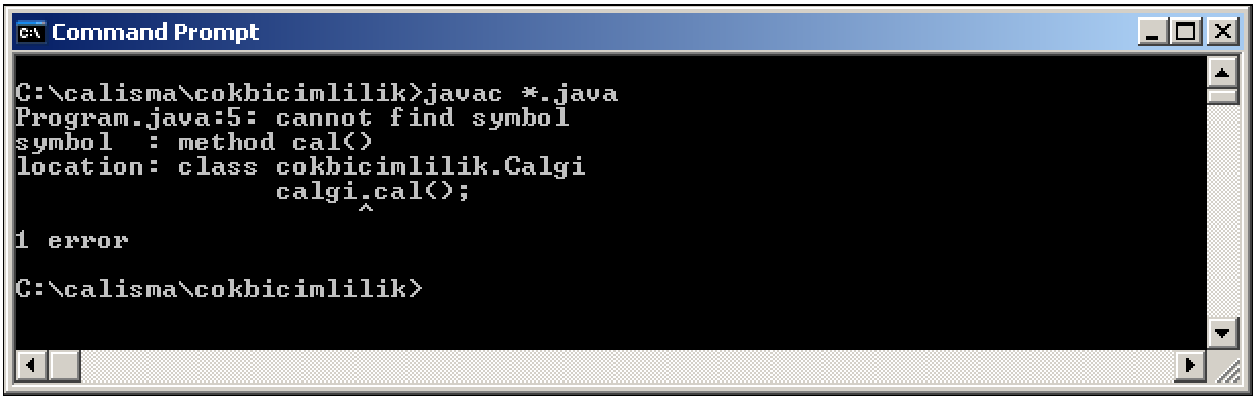
Yanıt:

Derleyici açısından baktığımızda, **Program** sınıfının içindeki

```
      calgi.cal();
```

satırında hata olduğunu söyleyebiliriz. Çünkü **calgi** referansı **Calgi** sınıfına aittir ve derleme zamanında bu referansın hangi nesneyi göstereceği belli değildir. Öyleyse **Gitar** sınıfında **cal()** yönteminin

bulunması ve çalışma zamanında **`Gitar`** nesnesi üzerinden **`cal()`** yönteminin işletilecek olması, derleyici açısından önemli değildir. Önemli olan **`Calgi`** sınıfında **`cal()`** yönteminin bulunmamasıdır. Referans değişkenin ait olduğu sınıfta ya da onun ata sınıflarından herhangi birisinde böyle bir yöntem olmadığı için derleyici bu yöntem çağrısına izin vermeyecektir:



5.6.4 Çokbiçimlilik Ne İşe Yarar?

Çokbiçimlilik, *ne* yapılacağı ile *nasıl* yapılacağını birbirinden ayırmaktadır. Çünkü her alt sınıf, ata sınıftaki bir yöntemi geçersiz kıldığında o yöntemi daha özel bir şekilde gerçekleştiriyor demektir. Dolayısıyla işin *nasıl* yapılacağı alt sınıfla birlikte değişmektedir. Ata sınıf tipinden referans üzerinden yapılan yöntem çağrısı ise *ne* yapılacağını belirler. calgi.cal(); deyimi, bir çalgının çalmasını istemektedir; *ne* yapılacağı bellidir. Ancak o çalgının ne olduğu ve *nasıl* çalacağı ancak çalışma zamanında belirlenmiş olacaktır.

Çokbiçimlilik uygulamaların genişletilebilirliğini sağlar. Derleme zamanında ne yapılacağının belli olması, o işin nasıl yapılacağının ise çalışma zamanında oluşturulan nesne ile birlikte belirlenmesi, geliştirilen sisteme yeni *nasıl*lar eklenmesini kolaylaştırır.

Çokbiçimlilik sayesinde, bir ata sınıfın sunduğu yöntemleri geçersiz kılan alt sınıflar yardımı ile ata sınıfa göre kodlanmış tek bir kod kesimine farklı davranışlar yüklemek olanaklı olmaktadır. Öyleyse, elimizde devingen olarak davranışı değiştirilebilen bir altyapı var demektir. Bu devingen

altyapıya yeni türlerin eklenmesi, kalıtım ve geçersiz kılma ilişkileri çerçevesinde oldukça kolaydır.

Çokbiçimliliğin ne işe yaradığını anlatmanın en kolay yolunun, çokbiçimlilik olmasaydı nasıl kod yazılması gerektiğini göstermek olduğunu düşünüyorum. Bu nedenle önce çokbiçimlilik mekanizmasının olmadığını varsayarak örnek bir program yazacağız. Ancak hemen öncesinde soyut sınıflardan bahsetmeliyiz. Çünkü kalıtım ve çokbiçimlilik kavramları soyut sınıflar olmadan düşünülemez.

> ***Çokbiçimlilik <u>ne</u> yapılacağı ile <u>nasıl</u> yapılacağını birbirinden ayırarak, kodun genişletilebilirliğini sağlar.***

5.6.5 Soyut Sınıflar (*Abstract Classes*)

Çalgılarla ilgili olarak gerçekleştirdiğimiz çokbiçimlilik örneğimize geri dönelim:

Bütün çalgılar için **`cal()`** yönteminin tanımlanmasının anlamlı olduğu açıktır. Öyleyse kalıtım ağacının en üstünde bulunan **`Calgi`** sınıfında **`cal()`** yöntemini kodlamalıyız. Peki bu yöntemin gerçekleştirimini (*implementation*) nasıl yapacağız?

> ***Bir yöntemin gerçekleştirimi, { işareti ile başlayıp } işareti ile biten kesimdir. Bu kesime "yöntemin gövdesi (method body)" de denir. Yöntemin döndüreceği değerin türü, yöntemin adı ve parametrelerini belirten kesime ise "yöntem bildirimi (method declaration)" adı verilir.***
>
> ***void yontem(int param1, String param2) → yöntemin bildirimi***
>
> ***{***
>
> ***....*** → ***yöntemin gövdesi / yöntemin gerçekleştirimi***
>
> ***}***

`Calgi` sınıfı bütün çalgıların ata sınıfıdır. Çalgı dendiği zaman herkesin aklında bir çalgı canlanmaktadır. Ancak **`Calgi`** sınıfında **`cal()`** yöntemini

kodlamak istediğimizde, gerçekçi bir gerçekleştirim yapamayız. Örneğin eğer bilgisayarın ses kartından ses verebilen bir gerçekleştirim yapıyor olsaydık, **`Calgi`** sınıfının içindeki **`cal()`** yönteminin ses kartından nasıl bir ses vermesi gerektiğine karar veremeyecektik. Çünkü çalma eylemi, çalgının türüne göre belli olacaktır. Bir gitar ile bir flütün aynı şekilde ses veremeyeceği düşünülürse, bunların ata sınıfı olan **`Calgi`** sınıfında kodlanacak yöntem gövdesinin aslında çalma eylemi ile ilgisi olmayan, dolayısıyla çok da anlamlı olmayan bir kod kesimi olacağı görülür.

Eğer bir sınıftaki herhangi bir yöntem için anlamlı bir gerçekleştirim tanımlanamıyorsa, o yöntem için gerçekleştirimin kodlanması zorunlu değildir. Sınıfın içinde yöntemin yalnızca bildirimi bulunabilir. Böyle tanımlanan yöntemlere soyut yöntem (*abstract method*) adı verilir ve bir yöntemin soyut olduğunu belirtmek için **`abstract`** anahtar sözcüğü kullanılır.

Eğer bir sınıfın içindeki yöntemlerden en az bir tanesi soyut yöntemse, sınıf da soyut tanımlanmalıdır.

Aşağıdaki örnekte **`Calgi`** sınıfının **`cal()`** yöntemi soyut tanımlanmış, dolayısıyla **`Calgi`** sınıfı da soyut olarak belirtilmiştir:

```
public abstract class Calgi {
      public void abstract cal();
}
```

Kod 5-90. abstract yöntem tanımlama

Bir sınıfın soyut olması, o sınıfın nesnesinin oluşturulamamasının derleyici tarafından denetlenmesini sağlar. Eğer kodun herhangi bir yerinde bir soyut sınıfın nesnesini oluşturmak üzere kod yazılmışsa, derleyici bu kod kesimine hata verecektir.

```
public class Program {
      public static void main(String[] args) {
            Calgi calgi = new Calgi();    // derleme hatası
      }
}
```

Kod 5-91. Soyut sınıfın nesnesinin oluşturulamaması

Soyut sınıf, nesnesi oluşturulamayan/oluşturulamaması gereken sınıftır. Adından da anlaşılacağı üzere soyut sınıf, soyut olarak kalmak üzere kodlanmış bir sınıftır. Nesne oluşturmak ise sınıfı somutlaştırmak olacaktır. Eğer soyut sınıfın nesnesi oluşturulabilseydi, o nesne üzerinden soyut tanımlanmış olan yöntemin de çağrılabilmesi olanaklı olurdu. Ancak yöntemin soyut tanımlanmasının nedeni zaten yönteme bir gerçekleştirimin kodlanamamış olmasıdır. Öyleyse, soyut sınıfın nesnesinin oluşturulabilmesi, sınıfın ve o sınıfın yöntem(ler)inin soyut tanımlanma gerekçesi ile çelişmektedir.

Soyut sınıf neden tanımlanır?

Soyut yöntemin gerçekleştirimi kodlanmadığına ve soyut sınıfın nesnesi oluşturulamadığına göre böyle bir sınıf ve sınıfa ait yöntem(ler) neden kodlanmaktadır?

Konunun başlangıcında da belirttiğimiz gibi soyut sınıf aslında kalıtım hiyerarşisinde yeri olan, modelin tasarımı için mantıksal gerekliliği bulunan bir sınıftır. Nesneye yönelik modeldeki bütün sınıfların nesnelerinin oluşturulması gibi bir zorunluluk yoktur. Önemli olan mantıksal olarak doğru bir model oluşturmak, sınıflar arasındaki ilişkileri doğru tanımlamaktır. Sınıfın soyut olması ise bir sonraki aşamada, sınıfın yöntemlerinin kodlanması sırasında ortaya çıkmaktadır. Çalgı örneğinde **`Calgi`** sınıfı gibi bir ata sınıfın olmaması düşünülemez. Ata sınıfta doğal olarak bütün alt sınıflarda ortak olan nitelik ve yöntemler bulunacaktır. Ancak bütün yöntemlerin gerçekleştirimlerinin kodlanması zorunlu olmadığı gibi olanaklı da olmayabilir.

Soyut sınıflar çokbiçimlilik ilkesinin gerçekleştirilmesinde önemli bir yere sahiptir. Geç bağlama ve kalıtım sayesinde, ata sınıf referansı üzerinden yapılan yöntem çağrılarının çokbiçimliliği meydana getirdiğini biliyoruz. Bir yöntemin soyut tanımlanması, o sınıfın alt sınıf(lar)ında bu yöntemin gerçekleştiriminin kodlanmasını, yani geçersiz kılınmasını zorlar. Çünkü eğer alt sınıf ata sınıftan aldığı soyut yöntemi geçersiz kılmıyorsa, o zaman alt sınıfta soyut bir yöntem var demektir. Bu durumda alt sınıfın da soyut

tanımlanması gerekecektir ve nesnesi oluşturulamaz. Yöntemin çağrılabilmesi için ise önce nesne oluşturulması gerekir (**`static`** yöntemler bu ifadenin dışındadır). Öyleyse eğer bir yöntem çağrılabiliyorsa, yöntemin içinde bulunduğu sınıfın nesnesi oluşturulmuş demektir. Nesne oluşturulabildiğine göre soyut tanımlanmış yöntem(ler) alt sınıfta geçersiz kılınmış olmalıdır. Böylece, ata sınıf referansı üzerinden soyut yöntem çağrısı yapılarak çokbiçimlilik gerçekleştirilmiş olur.

Çokbiçimliliğin ise kodun genişletilebilirliğine nasıl katkıda bulunduğundan bahsetmiştik. Soyut sınıflar, çokbiçimliliği zorlayarak kodun genişletilebilirliğini arttırmak üzere kullanılabilirler. Genellikle bir tasarımın genişletme noktaları olarak düşünülen kısımları soyut yöntemler olarak tanımlanır ve bu tasarıma eklenecek olan sınıfların, soyut yöntemleri içeren sınıfın alt sınıfı olarak kodlanmaları ve soyut yöntemleri geçersiz kılmaları beklenir. Böylece, ata sınıf referansları ile çalışacak şekilde tasarlanan uygulamanın, soyut tanımlanan yöntemler için alt sınıflarda anlamlı gerçekleştirimler bulması garanti edilmiş olur.

Örnek Senaryo:

Bir giyim mağazasında çalışan kişilerin maaşlarını ekrana yazdıracak bir program geliştireceğiz. Çalışanların maaşları, görevlerine göre farklı şekillerde hesaplanıyor:

Müdür: Aylık 2.000 YTL maaş + prim ile çalışıyor. Prim ise toplam satış tutarına göre hesaplanıyor. O ayki tutarın 20.000 TL'yi geçmesi durumunda, 20.000 TL'den fazla olan miktarın %10'u kadar prim alıyor.

Tezgahtar: Prim ile çalışıyor. Yaptığı aylık toplam satışın %10'unu alıyor.

İdari Personel: Aylık 800 TL sabit maaş ile çalışıyor.

Hizmetli: Saatlik ücret ile çalışıyor. Saat ücreti 10 TL.

Mağazada 1 müdür, 2 tezgahtar, 2 idari personel ve 2 hizmetli çalışıyor. Ay sonunda kimin ne kadar satış yaptığı ya da kaç saat çalıştığı sisteme girilecek ve herkesin alması gereken maaş, kişinin yaptığı iş, adı ve soyadı ile birlikte ekranda görülecek.

Not: Çalışanlar ile ilgili bilgileri kullanıcının girmesi gerekmiyor. Bu bilgileri örnek programın içinde kodlayabiliriz. Amaç bu programı bir mağazada kullanmak değil, yalnızca bazı kavramları kullanarak tasarım yapmak.

<u>1. Çözüm</u>: Her farklı çalışan türü için ayrı bir sınıf ve bu sınıfların kullanılışını örnekleyen küçük bir program örneği kodlayalım:

```
package cokbicimlilik.soyutsiniflar.magaza.cozum1;
public class Mudur {
      private String ad;
      private String soyad;
      private int maas;

      public Mudur(String ad, String soyad) {
            this.ad = ad;
            this.soyad = soyad;
      }
      public String getAd() {
            return ad;
      }
      public String getSoyad() {
            return soyad;
      }
      public void setMaas(int maas) {
            this.maas = maas;
      }
      public String getBilgi() {
            return "Müdür: " + this.ad + " " + this.soyad +
                   ", bu ay " + this.maas + " YTL alacaktır.";
      }
}}
```

Kod 5-92. Soyut sınıf örneği, 1. çözüm – Mudur sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum1;
public class Hizmetli {


      private String ad;
      private String soyad;
      private int maas;

      public Hizmetli(String ad, String soyad) {
```

```
            this.ad = ad;
            this.soyad = soyad;
      }
      public String getAd() {
            return ad;
      }
      public String getSoyad() {
            return soyad;
      }
      public void setMaas(int maas) {
            this.maas = maas;
      }
      public String getBilgi() {
            return "Hizmetli: " + this.ad + " " + this.soyad +
                  ", bu ay " + this.maas + " YTL alacaktır.";
      }
}
```

Kod 5-93. Soyut sınıf örneği, 1. çözüm – Hizmetli sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum1;
public class Tezgahtar {
      private String ad;
      private String soyad;
      private int maas;

      public Tezgahtar(String ad, String soyad) {
            this.ad = ad;
            this.soyad = soyad;
      }
      public String getAd() {
            return ad;
      }
      public String getSoyad() {
            return soyad;
      }
      public void setMaas(int maas) {
            this.maas = maas;
      }
      public String getBilgi() {
            return "Tezgahtar: " + this.ad + " " + this.soyad +
                  ", bu ay " + this.maas + " YTL alacaktır.";
```

```
	}
}
```

Kod 5-94. Soyut sınıf örneği, 1. çözüm – Tezgahtar sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum1;
public class IdariPersonel {
	private String ad;
	private String soyad;
	private int maas;

	public IdariPersonel(String ad, String soyad) {
		this.ad = ad;
		this.soyad = soyad;
	}

	public String getAd() {
		return ad;
	}
	public String getSoyad() {
		return soyad;
	}
	public void setMaas(int maas) {
		this.maas = maas;
	}
	public String getBilgi() {
		return "İdari Personel:" + this.ad + " " + this.soyad +
				", bu ay " + this.maas + " YTL alacaktır.";
	}
}
```

Kod 5-95. Soyut sınıf örneği, 1. çözüm – IdariPersonel sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum1;
public class Magaza {
	private final static int MUDUR_CIPLAK_MAASI = 2000;
	private final static int MUDUR_PRIM_LIMITI = 20000;
	private final static int HIZMETLI_SAATLIK_UCRET= 10;
	private final static int IDARI_PERSONEL_MAASI = 800;

	public static void main(String[] args) {

		Hizmetli[] hizmetliler = new Hizmetli[2];
```

```
            hizmetliler[0] = new Hizmetli("Ali", "Kaya");
            hizmetliler[0].setMaas(HIZMETLI_SAATLIK_UCRET * 60);
            hizmetliler[1] = new Hizmetli("Ahmet", "Ateş");
            hizmetliler[1].setMaas(HIZMETLI_SAATLIK_UCRET * 40);

            IdariPersonel[] idariPersoneller = new IdariPersonel[2];
            idariPersoneller[0] = new IdariPersonel("Ayşe", "Demir");
            idariPersoneller[0].setMaas(IDARI_PERSONEL_MAASI);
            idariPersoneller[1] = new IdariPersonel("Mehmet", "Çelik");
            idariPersoneller[1].setMaas(IDARI_PERSONEL_MAASI);

            Tezgahtar[] tezgahtarlar = new Tezgahtar[2];
            tezgahtarlar[0] = new Tezgahtar("Okan", "Yeşil");
            tezgahtarlar[0].setMaas(15000 / 10);
            tezgahtarlar[1] = new Tezgahtar("Burcu", "Seğmen");
            tezgahtarlar[1].setMaas(22000 / 10);

            Mudur mudur = new Mudur("Furkan", "Kartal");
            mudur.setMaas((37000 - MUDUR_PRIM_LIMITI) / 10 +
                          MUDUR_CIPLAK_MAASI);



            System.out.println("Çalışanların maaşları:");
            System.out.println(mudur.getBilgi());
            for (int i = 0; i < tezgahtarlar.length; ++i) {
                  System.out.println(tezgahtarlar[i].getBilgi());
            }
            for (int i = 0; i < idariPersoneller.length; ++i) {
                  System.out.println(idariPersoneller[i].getBilgi());
            }
            for (int i = 0; i < hizmetliler.length; ++i) {
                  System.out.println(hizmetliler[i].getBilgi());
            }
      }
}
```

Kod 5-96. Soyut sınıf örneği, 1. çözüm – Magaza sınıfı

```
Çalışanların maaşları:
Müdür: Furkan Kartal, bu ay 3700 YTL alacaktır.
Tezgahtar:Okan Yeşil, bu ay 1500 YTL alacaktır.
Tezgahtar:Burcu Seğmen, bu ay 2200 YTL alacaktır.
```

```
İdari Personel:Ayşe Demir, bu ay 800 YTL alacaktır.
İdari Personel:Mehmet Çelik, bu ay 800 YTL alacaktır.
Hizmetli:Ali Kaya, bu ay 600 YTL alacaktır.
Hizmetli:Ahmet Ateş, bu ay 400 YTL alacaktır.
```

Çıktı 5-12. Soyut sınıf örneği, 1. çözüm – Programın çıktısı

Şimdi bu çözümü değerlendirmeye çalışalım:

1- **Mudur**, **IdariPersonel**, **Tezgahtar** ve **Hizmetli** sınıfları incelenirse, bu sınıflardaki kodun çoğunun aynı işi yaptığı ancak her sınıfta yeniden kodlandığı görülür. Bunun yerine bütün bu sınıflar **Calisan** gibi bir ata sınıftan kalıtım ilişkisi ile hem nitelikleri hem de o nitelikler üzerinde işlem yapan yöntemleri alabilirlerdi.

> ***Kod tekrarı istenen birşey değildir. Çünkü kod tekrarı kodun bakım kolaylığını kötü yönde etkiler. Herhangi bir değişiklik yapılması söz konusu olduğunda, kodun bütün tekrarlarının elden geçirilmesi gerekir. Bu da doğal olarak kodun bakım maliyetini arttıracağı gibi eksik kalan, unutulan değişiklikler sonucunda tutarsız ve hata içeren programların ortaya çıkması olasılığı artar. Kodu tekrarlamamak için yapılabileceklerden birincisi, aynı işi yapan kod kesimlerinin işlevler içerisine koyulup, kodun kopyalanması yerine işlevin çağrılmasıdır. Nesneye yönelik programlamada ise kalıtım ya da içerme ilişkileri ile kodun yeniden kullanılması sağlanabilir.***

2- **Magaza** sınıfında, bütün çalışanlar için ayrı ayrı diziler tanımlandığı görülmektedir. Her dizi belli bir tipteki çalışanları tutmaktadır. Çalışanların maaşlarının ekrana dökülmesi için de bu diziler üzerinden ayrı ayrı geçilmesi gerekmektedir. Şimdi bu koda **MudurYardimcisi** sınıfını eklediğimizi ve onun da maaşının farklı bir şekilde hesaplandığını düşünelim. Bu durumda **Magaza** sınıfında **MudurYardimcisi[]** şeklinde yeni bir dizi tanımlamamız, bu dizinin üzerinden geçecek yeni bir kod kesimi daha yazmamız gerekecektir. Oysa eğer 1. maddede bahsedilen kalıtım ilişkisi kurulsaydı, bütün çalışanlar ata sınıf türünden tanımlanmış bir dizide (**Calisan[]**) saklanabilirdi. Calisan sınıfına **maasHesapla()** gibi bir yöntem tanımlanır ve maaşları ekrana döken kesim bu dizi üzerinden

çalışan tek bir döngü olabilirdi. Böylece **MudurYardimcisi** gibi yeni bir sınıf eklendiğinde kodda herhangi bir değişiklik yapılması gerekmezdi.

3- **Magaza** sınıfı, bütün çalışan türlerinin maaşlarını hesaplama bilgisinin kodlandığı yer olarak görülüyor. Oysa her sınıf kendi sınıfına özel maaş hesaplama işlemini yaparsa, örneğin bir müdürün maaşının hesaplanışı **Mudur** sınıfı içinde kodlanırsa, kodun daha kolay yönetilebilmesi sağlanır. Bir **MudurYardimcisi** sınıfı eklendiğinde, bu sınıfa özel maaş hesaplama bilgisinin de **Magaza** sınıfına kodlanacak olması güzel bir yaklaşım olmayacaktır.

2. Çözüm: Şimdi bu bilgiler ışığında kalıtım ilişkisini de kurarak sınıfları yeniden kodlayalım:

Öncelikle **Calisan** adlı bir ata sınıf tanımlayarak, bütün alt sınıflarda bulunan kod kesimlerini bu sınıfın içine alacağız. 1. çözüm ile ilgili değerlendirmemizin 2. ve 3. maddelerinde belirttiğimiz gibi maaş hesaplama işlemlerini de her sınıfın kendisine özel yapacağız. Bu durumda, **Calisan** adlı ata sınıfa, bütün alt sınıflarda geçersiz kılınmasını istediğimiz **maasHesapla()** yöntemini tanımlayacağız. Ancak bu yöntemin gerçekleştirimini (yöntem gövdesini) nasıl ve neye göre kodlamalıyız? **Calisan** sınıfı bütün çalışanların ata sınıfı olduğuna ve çalışanın türü özelleşmediği (alt sınıf) sürece maaşının nasıl hesaplanacağı bilinmediğine göre, **Calisan** sınıfındaki **maasHesapla()** yönteminin gerçekleştirimini yapamayacağız (ya da hiçbirşey yapmayan, içi boş bir yöntem gövdesi yazacağız ancak anlamlı olmayacak!). Öyleyse **maasHesapla()** yöntemini soyut olarak tanımlamalıyız. Bu durumda **Calisan** sınıfı da soyut olacaktır.

```
package cokbicimlilik.soyutsiniflar.magaza.cozum2;
public abstract class Calisan {
      private String ad;
      private String soyAd;

      public Calisan(String ad, String soyAd) {
            this.ad = ad;
            this.soyAd = soyAd;
```

```
	}
	public String getAd() {
		return ad;
	}
	public String getSoyAd() {
		return soyAd;
	}
	public abstract int maasHesapla();

	public String getBilgi() {
		return this.kimimBen() + ": " + this.ad + " " + this.soyAd +
			", bu ay " + this.maasHesapla() + " YTL alacaktır.";
	}
	public abstract String kimimBen();
}
```

Kod 5-97. Soyut sınıf örneği, 2. çözüm – Calisan sınıfı

Bütün çalışan türleri için ortak olan **`ad`** ve **`soyad`** nitelikleri, bunların **`get`** yöntemleri ve **`maasHesapla()`** yöntemini **`Calisan`** sınıfının içinde kodladık. Ayrıca yine **`Calisan`** sınıfında bir de **`getBilgi()`** yöntemi olduğu görülüyor.

Önceki çözüm incelenirse, bütün çalışan türleri için ortak olan bir başka yöntemin de **`getBilgi()`** yöntemi olduğu görülür. Ancak **`getBilgi()`** yöntemini ata sınıfa kodlarsak, çalışanın müdür mü, tezgahtar mı yoksa hizmetli mi olduğunu nasıl ayırdedeceğimiz sorunu ortaya çıkar. İşte bu sorunu çözebilmek için **`Calisan`** sınıfına soyut bir yöntem daha tanımladık: **`kimimBen()`** yöntemi. Bu yöntem soyut tanımlandı çünkü hem ata sınıfta bu yöntemin kodlanması için gerekli ve yeterli bilgi yoktur, hem de bu yöntemin alt sınıflarda mutlaka geçersiz kılınması gereklidir. Böylece **`getBilgi()`** yöntemi işletilirken, çalışma zamanında hangi nesne üzerinden çağrılmışsa o nesnenin ne tipte bir çalışana ait olduğunu döndürebilecektir.

```
package cokbicimlilik.soyutsiniflar.magaza.cozum2;
public class Mudur extends Calisan  {
	private static final int CIPLAK_MAAS = 2000;
	private static final int CIPLAK_MAAS = 2000;
```

```
	private static final int PRIM_LIMITI = 20000;

	private int satis;

	public Mudur(String ad, String soyad){
		super(ad, soyad);
	}

	public void setToplamSatis(int satis) {
		this.satis = satis;
	}

	public int maasHesapla() {
		if (this.satis > Mudur.PRIM_LIMITI) {
			return CIPLAK_MAAS +
				(this.satis - Mudur.PRIM_LIMITI) / 10;		}
		return CIPLAK_MAAS;
	}

	public String kimimBen() {
		return "Müdür";
	}
}
```

Kod 5-98. Soyut sınıf örneği, 2. çözüm – Mudur sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum2;
public class Hizmetli extends Calisan {
	private static final int SAAT_UCRETI = 10;
	private int mesaiSaati;

	public Hizmetli(String ad, String soyad){
		super(ad, soyad);
	}

	public void setMesaiSaati(int mesaiSaati) {
		this.mesaiSaati = mesaiSaati;
	}

	public int maasHesapla() {
		return this.mesaiSaati * Hizmetli.SAAT_UCRETI;
	}
```

```
	public String kimimBen() {
		return "Hizmetli";
	}
}
```

Kod 5-99. Soyut sınıf örneği, 2. çözüm – Hizmetli sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum2;
public class Tezgahtar extends Calisan {
	private int satis;

	public Tezgahtar(String ad, String soyad){
		super(ad, soyad);
	}

	public void setSatis(int satis) {
		this.satis = satis;
	}

	public int maasHesapla() {
		return this.satis / 10;
	}

	public String kimimBen() {
		return "Tezgahtar";
	}
}
```

Kod 5-100. Soyut sınıf örneği, 2. çözüm – Tezgahtar sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum2;
public class IdariPersonel extends Calisan {
	private static final int MAAS = 800;

	public IdariPersonel(String ad, String soyad){
		super(ad, soyad);
	}

	public int maasHesapla() {
		return IdariPersonel.MAAS;
	}

```

```
      public String kimimBen() {
            return "İdari Personel";
      }
}
```

Kod 5-101. Soyut sınıf örneği, 2. çözüm – IdariPersonel sınıfı

```
package cokbicimlilik.soyutsiniflar.magaza.cozum2;
public class Magaza {

      public static void main(String[] args) {
            Calisan calisanlar[] = new Calisan[7];

            calisanlar[0] = new Mudur("Furkan", "Kartal");
            ((Mudur)calisanlar[0]).setToplamSatis(37000);

            calisanlar[1] = new Tezgahtar("Okan", "Yeşil");
            ((Tezgahtar) calisanlar[1]).setSatis(15000);
            calisanlar[2] = new Tezgahtar("Burcu", "Seğmen");
            ((Tezgahtar) calisanlar[2]).setSatis(22000);

            calisanlar[3] = new IdariPersonel("Ayşe", "Demir");
            calisanlar[4] = new IdariPersonel("Mehmet", "Çelik");

            calisanlar[5] = new Hizmetli("Ali", "Kaya");
            ((Hizmetli) calisanlar[5]).setMesaiSaati(60);
            calisanlar[6] = new Hizmetli("Ahmet", "Ateş");
            ((Hizmetli) calisanlar[6]).setMesaiSaati(40);

            System.out.println("Çalışanların maaşları:");
            for (int i = 0; i < calisanlar.length; ++i) {
                  System.out.println(calisanlar[i].getBilgi());
            }
      }
}
```

Kod 5-102. Soyut sınıf örneği, 2. çözüm – Magaza sınıfı

```
Çalışanların maaşları:
Müdür: Furkan Kartal, bu ay 3700 YTL alacaktır.
Tezgahtar: Okan Yeşil, bu ay 1500 YTL alacaktır.
Tezgahtar: Burcu Seğmen, bu ay 2200 YTL alacaktır.
İdari Personel: Ayşe Demir, bu ay 800 YTL alacaktır.
```

```
İdari Personel: Mehmet Çelik, bu ay 800 YTL alacaktır.
Hizmetli: Ali Kaya, bu ay 600 YTL alacaktır.
Hizmetli: Ahmet Ateş, bu ay 400 YTL alacaktır.
```

Çıktı 5-13. Soyut sınıf örneği, 2. çözüm – Programın çıktısı

Görüldüğü gibi 2. çözüm de aynı çıktıyı üretmektedir. Buna ek olarak, bu yeni çözüme **`MudurYardimcisi`** gibi yeni bir tür ekleyebilmek için yapmamız gereken, bu sınıfı **`Calisan`** sınıfından kalıtım ilişkisi ile özelleştirmek ve ata sınıfta soyut tanımlanmış yöntemleri geçersiz kılmaktan ibarettir. Maaşların ekrana döküldüğü kod kesiminde başka herhangi bir değişiklik yapmak gerekmeyecektir. Bunu sağlayan araç, kalıtım ve geç bağlama ile gerçekleştirilen çokbiçimlilik ve çokbiçimliliği gerçekleştirecek olan yöntemlerin geçersiz kılınmalarını zorlayan soyut yöntem ve soyut sınıf tanımlarıdır.

5.6.6 Arayüzler (*Interfaces*)

Arayüz denilince genelde herkesin aklına ilk önce kullanıcı arayüzleri gelir (*GUI - Graphical User Interface*). Örneğin Windows işletim sistemi kullanıyorsanız, bilgisayarınızı açtığınızda ekranınızda gördükleriniz Windows'un kullanıcı arayüzüdür. Bu arayüz, fare ve klavye çevre birimlerini de kullanarak Windows ile etkileşimde bulunabilmenizi sağlar.

Nesneye yönelik programlama bağlamında ise arayüz, kavramsal olarak kullanıcı arayüzü ile benzer "iş"i yapan bir araç olarak değerlendirilmelidir. Önce bu "iş"i ve arayüzün en genel ifade ile ne olduğunu açıklamaya çalışalım. Sonra da Java'da arayüzlerin nasıl kodlandığına ve kullanımlarına bakalım.

Arayüz Kavramı

Arayüz temelde bir sistemin/alt sistemin başka sistem/alt sistemlerle arasındaki iletişimin nasıl olacağını belirler. Bu, sözkonusu olan arayüz ister kullanıcı arayüzü olsun, ister iki donanım arasındaki arayüz olsun mantıksal olarak aynıdır.

Bir programın kullanıcı arayüzü, kullanıcıların o programdan nasıl hizmet alabileceğini ve ne yapabileceklerini belirler. Burada program bir sistem, kullanıcı ise o sistemden hizmet alan başka bir sistemdir. Kullanıcı programın hangi dille kodlandığını, kullandığı algoritmaları ya da veri yapılarını bilmez. Kullanıcı sadece fare ile ekranın neresine tıkladığında ne olacağını bilir. Başka bir deyişle kullanıcı arayüzü bilir, tanır; ancak arayüzün arkasındaki ayrıntıları bilemez, bilmesine de gerek yoktur.

Ekrandan gelen görüntü kablosunun bilgisayara bağlandığı yuva, bilgisayarın arayüzlerinden birisidir. Bu arayüz sayesinde, görüntü verebilecek herhangi bir donanım bilgisayara bağlanabilir. Örneğin ekran yerine bir yansıtıcı ile bilgisayardan gelen görüntü bir perdeye aktarılabilir. Önemli olan o arayüzü tanıyan/bilen bir donanım olmasıdır. Bilgisayar görüntüyü üreten sistem, bilgisayara bağlanan cihaz ise o görüntüyü bir şekilde gösterebilen başka bir sistemdir. Her iki sistem de birbirlerinin iç yapılarını bilmek durumunda değildir. Bilgisayar için görüntünün perdede, LCD ekranda ya da CRT ekranda yansıtılmasının bir farkı olmadığı gibi, bir ekranın da görüntüyü nasıl bir bilgisayardan aldığının bir önemi yoktur. Burada arayüz iki sistem arasında iletişimin nasıl olacağını belirleyen bir sözleşmedir. Kablonun takıldığı yuvadaki hangi iğnenin hangi anlama geleceği bellidir ve bu arayüze uygun olarak üretilmiş çok farklı donanımlar bulunabilir.

Kısaca "arayüz bir sistemin başka sistemlerle iletişim kurmak/birlikte çalışmak için sunduğu bir sözleşme, bir bütünleşme noktasıdır" denilebilir. Eğer bir sistemi kullanmak istiyorsak o sistemin arayüzünü, arayüzünde sağlanan yetenekleri uygun şekilde kullanmayı öğrenmemiz gerekir. Bilgisayarın görüntü aktarmak için sunduğu arayüz, aslında herbirinin bir anlamı olan elektrik imleri ileten iğnelerden oluşmaktadır. Bir cihazı görüntü almak üzere bilgisayara bağlayabilmemiz için bu imleri alabilmesi, aldığı imleri bir şekilde görüntüye dönüştürmesi yeterlidir. Bu dönüştürmeyi nasıl yaptığıyla ilgilenmeyiz. Bu yaklaşımla, eğer arayüzün

gereğini yerine getiriyorsa bir cep telefonunu da bilgisayarın ekranı olarak kullanma şansımız vardır.

Java'da Arayüzler

Arayüzler programlama bağlamında da sistemler arasında bütünleşme noktaları tanımlamak için kullanılırlar. Buradaki "sistemler" programın bir modülünü/alt sistemini oluşturan sınıflara karşılık gelmektedir.

Bir arayüz, bir grup yöntem bildiriminden (*method declaration*) oluşan bir sözleşmedir. Bu yöntemler, arayüzü tanımlayan sistemin, kendisiyle birlikte çalışmak isteyen diğer alt sistem(ler)in gerçekleştirmesini (*implement*) beklediği yöntemlerdir. Başka bir deyişle, arayüzü tanımlayan sistem, birlikte çalışabileceği diğer sistemlerin neye benzemeleri gerektiğini belirtmektedir. Arayüz bir sözleşme olduğuna göre, bu arayüz üzerinden sistemle bütünleşen diğer sistem, sözleşmedeki bütün maddeleri yerine getirmeli yani arayüzde bulunan bütün yöntemlerin gerçekleştirimlerini (*method implementation*) sağlamalıdır.

Bunu bir örnekle açıklamaya çalışalım:

Bir çizim programı geliştirdiğimizi ve ekrandaki nesneleri yönetecek bir sınıf yazmak istediğimizi düşünelim. Ancak yazacağımız sınıfı öyle tasarlamalıyız ki, sonradan eklenen herhangi bir sınıfın nesneleri ile de çalışabilmeli.

Bu gereksinimleri karşılamak üzere, ekrandaki nesnelerin bir listesini tutacak ve ekrandaki görüntüyü yenilemesi gerektikçe bu listedeki bütün nesnelere **`ciz()`** iletisini gönderecek bir sınıf kodlayabiliriz:

```
public class SekilYoneticisi {
      public static void main(String[] args) {
            Sekil sekiller[];

            // ......

            // şekil nesnelerinin her birine ciz() iletisi gönder
            for (int i = 0; i < sekiller.length; ++i) {
                  sekiller[i].ciz();
```

```
            }

            // ......
        }
}
```

Kod 5-103. SekilYoneticisi sınıfı

Tasarımımızın sonradan eklenecek sınıflarla birlikte çalışabilmesi için de aşağıdaki gibi bir ata sınıf tanımlayıp, ekranda çizilmesi istenen bütün nesnelerin bu sınıfın alt sınıfı olmasını zorlayabiliriz:

```
public abstract class Sekil {
      public abstract void ciz();
}
```

Kod 5-104. Sekil sınıfı

Bu tasarımın çözemeyeceği iki önemli sorun bulunmaktadır:

Birinci sorun, sisteme eklenmesi istenen her sınıfın **Sekil** sınıfının alt sınıfı olması zorunluluğudur. Kalıtım ilişkisi "*is a*" ilişkisi olduğuna göre, **Sekil** sınıfının alt sınıfı olması istenen sınıfların gerçekten de birer şekil olmaları, modelin doğruluğu için bu ilişkinin de mantıksal açıdan doğru olması gerekmektedir.

İkinci sorun ise birinci sorun olarak ifade edilen mantıksal zorlamanın her zaman olanaklı olmamasından kaynaklanır. Çünkü ekranda çizmek istediğimiz bir sınıf eğer başka bir sınıfın alt sınıfı ise **Sekil** sınıfının alt sınıfı olamayacaktır. Dolayısıyla mantıksal ilişki yapay bir şekilde oluşturularak birinci sorun aşılsa bile ikinci sorun aşılamayacaktır.

Öte yandan, aslında herhangi bir sınıfın nesnesinin ekrana çizilebilmesi için **Sekil** sınıfının alt sınıfı olması değil, **Sekil** sınıfındaki **ciz()** yöntemine sahip olması gerekmektedir. Zaten **ciz()** yönteminin **abstract** tanımlanmasının nedeni de budur. Öyleyse yeni tanımlanan bir sınıfı **SekilYoneticisi** ile kullanabilmemiz için bize esas gerekli olan, bu sınıfın **ciz()** yönteminin gerçekleştirimini yaptığını garanti etmesidir. Başka bir deyişle, **SekilYoneticisi** ile birlikte çalışmak isteyen sınıfların, **ciz()**

yönteminden oluşan bir sözleşmeye uymaları gerekir. Bu sözleşme ise bir arayüz olarak tanımlanacaktır.

Java'da arayüz tanımı **`interface`** anahtar sözcüğü ile yapılır. Aşağıda **`Sekil`** arayüzünün tanımı görülmektedir:

```
public interface Sekil {
      void ciz();
}
```

Kod 5-105. Sekil.java

Herhangi bir sınıfın bir arayüz ile belirlenen sözleşmeye uyduğunu, yani o arayüzdeki yöntemlerin gerçekleştirimlerini sağladığını ifade etmek için ise **`implements`** anahtar sözcüğü kullanılır. Aşağıda, **`Sekil`** arayüzünün gerçekleştirimini yapan **`A`** sınıfı görülmektedir:

```
public class A implements Sekil {
      public void ciz() {
            // buraya A nesnelerini ekrana çizen kod yazılacak
      }
}
```

Kod 5-106. Sekil arayüzünü gerçekleştiren A sınıfı

Bir sınıf ancak tek bir sınıftan özelleştirilebilir. Yani **`extends`** anahtar sözcüğünden sonra yalnızca tek bir sınıfın adı yazılabilir ve o sınıfın alt sınıfı olunabilir. Ancak bir sınıfın hangi arayüzleri gerçekleştireceği konusunda bir sınır yoktur. Bir sınıf tek bir sınıfın alt sınıfı olabilirken, birçok arayüzü gerçekleştirebillir. Kalıtımın olması ya da olmamasının arayüz gerçekleştirilmesi ile ilgisi yoktur.

Aşağıdaki ömekte **`Kisi`** sınıfının alt sınıfı olan **`Ogrenci`** sınıfının **`Sekil`** arayüzünü gerçekleştirişi görülmektedir:

```
public class Ogrenci extends Kisi implements Sekil {
      // diğer kodlar
      public void ciz() {
            // buraya Ogrenci nesnelerini ekrana çizen kod yazılacak
      }
}
```

Kod 5-107. Ogrenci.java – Kisi sınıfının alt sınıfı, Sekil arayüzünü gerçekleştirir

Bir sınıfın hangi sınıfın alt sınıfı olduğuna bağlı olmaksızın herhangi bir arayüzü gerçeleştirebiliyor olması, yukarıda bahsettiğimiz sorunları ortadan kaldırmaktadır. Bir şekil olmadığı ve **Kisi** adlı bir sınıfın alt sınıfı olduğu halde, **SekilYoneticisi** sınıfı ile bütünleştirilebilmesi ve ekrana nesnelerinin çizilebilmesi için **Ogrenci** sınıfının yapması gereken tek şey **Sekil** arayüzünü gerçekleştirmektir.

Sekil arayüzünü gerçekleştiren sınıflar **ciz()** yöntemini içereceklerine göre, artık nesneleri ekrana çizen **SekilYoneticisi** sınıfı **ciz()** arayüzünü gerçekleştiren bütün sınıflarla çalışabilir.

```
public class SekilYoneticisi {
      public static void main(String[] args) {
            Sekil sekiller[];

            // ......

            // şekil nesnelerinin her birine ciz() iletisi gönder
            for (int i = 0; i < sekiller.length; ++i) {
                  sekiller[i].ciz();
            }

            // ......
      }
}
```

Kod 5-108. SekilYoneticisi.java

SekilYoneticisi sınıfının koduna tekrar bakalım:

Sekil tipinden bir dizi ve dizinin herbir elemanına yönlendirilen **ciz()** iletisini görüyoruz. Kodu derleyici açısından değerlendirdiğimizde, bu kodun derleme hatası vermemesi için **Sekil** türünün **ciz()** yöntemini içermesi gerektiğini söyleyebiliriz. **Sekil** türü ister sınıf ister arayüz olarak kodlanmış olsun, **ciz()** yöntemini içerdiğine göre herhangi bir derleme hatası söz konusu olmayacaktır.

Çalışma zamanında ise **sekiller[]** dizisinin **i** indisli elemanının gösterdiği nesne hangi nesne ise, **ciz()** yöntemi geç bağlama sayesinde o nesne

üzerinden işletilecektir. Eğer bu nesne **`ciz()`** yönteminin gerçekleştirimini sağlamıyorsa çalışma zamanında işletilecek yöntem bulunamayacak ve bir hata oluşacaktır. Ancak **`sekiller[]`** dizisindeki bütün nesneler **`Sekil`** türünden olduğuna göre, herbirisi **`ciz()`** yönteminin gerçekleştirimini sağlayacaktır. Çünkü **`Sekil`** arayüzü bir sözleşmedir ve **`ciz()`** yönteminin gerçekleştiriminin bulunacağını garanti etmektedir.

Arayüzler ve Soyut Sınıflar

Bir arayüzün içindeki bütün yöntemler gizliden (*implicitly*) **`public`** ve **`abstract`** tanımlıdır. Bu açıdan bakıldığında, bütün yöntemleri **`public`** ve **`abstract`** olan bir soyut sınıfın arayüz gibi görüleceği söylenebilir. Derleme kuralları ya da çalışma zamanındaki davranış açısından da soyut sınıflar ile arayüzler birbirlerine çok benzerler.

Örneğin bir soyut sınıfın alt sınıfı, ata sınıfından aldığı soyut yöntemlerin gerçekleştirimlerini sağlayarak bu yöntemleri geçersiz kılmazsa, alt sınıfın da soyut tanımlanması gerekir. Çünkü geçersiz kılmadığı yöntemler nedeniyle soyut yöntem içerecektir. Benzer şekilde, bir arayüzü gerçekleştirdiğini belirten bir sınıf, o arayüzdeki bütün yöntemlerin gerçekleştirimlerini içermezse soyut tanımlanmak zorundadır. Çünkü arayüzdeki yöntemler gizliden soyut yöntemlerdir ve o arayüzü gerçekleştiren sınıf soyut yöntem içeriyor demektir.

Soyut sınıflar çokbiçimliliğin gerçekleştirilmesinde önemli bir yere sahiptir. Soyut sınıflar sayesinde, soyut tanımlanan yöntemlerin alt sınıfta geçersiz kılınmaları zorlanır. Benzer şekilde arayüzler de içlerine tanımlanan yöntemlerin gerçekleştirimlerinin yapılmasını zorlarlar.

Bu tip kurallar, rahatlıkla öğrenilebilecek kurallardır ve herhangi bir uygulama geliştirme aracı yardımı ile kolayca düzeltilebilecek derleme hataları ile farkedilebilirler. Önemli olan, soyut sınıflar ile arayüzler arasındaki kavramsal farkların öğrenilmesidir.

Öncelikle, soyut sınıfın bütün yöntemlerinin soyut olması gerekmez. Soyut sınıflar nitelikler ve/ya da soyut olmayan yöntemler içerebilirler. Bu sınıflar

bazı özel yöntemlerin alt sınıflarda geçersiz kılınmalarını sağlamak üzere soyut tanımlanmışlardır. Buna güzel bir örnek, soyut sınıflar ile ilgili **`Magaza`** örneğimizdeki **`Calisan`** sınıfıdır. **`Calisan`** sınıfının nitelikleri ve soyut olmayan yöntemleri olmasına rağmen soyut tanımlanmasının nedeni, iki yönteminin alt sınıflarda o sınıflara özel bir şekilde gerçekleştirilmesinin zorlanmasını sağlamaktır. Bu açıdan bakıldığında, soyut sınıflar bir kalıtım sıradüzeninin parçasıdırlar ve bütün alt sınıfları "is a" ilişkisini sağlamalıdır.

Arayüzler ise nitelik ya da soyut olmayan yöntemler içeremezler. Bir arayüzün içinde ancak sabitler ve/ya da yöntemler bulunabilir. Sabitler, gizliden **`public static final`** tanımlı, yöntemler ise gizliden **`public abstract`** tanımlıdırlar. Bu açıdan soyut sınıflardan farklılaşırlar.

Kalıtım ilişkisi açısından bakıldığında, bir arayüzü gerçekleştiren sınıfların "is a" ilişkisini sağlamaları gerektiği söylenemez. Bir sınıf, bir arayüzü gerçekleştirdiğinde o arayüzdeki bütün "yöntem tanımlarını" kalıtımla almış olur. Ancak bu daha çok bütünleştirme amacı ile yapılır. Örneğin **`Ogrenci`** sınıfı **`Sekil`** arayüzünü gerçekleştirdiğinde, aslında mantıksal olarak "is a" illişkisini sağlamamakta, bununla birlikte **`ciz()`** yöntemini alarak nesneleri çizilebilir bir sınıf konumuna gelmektedir.

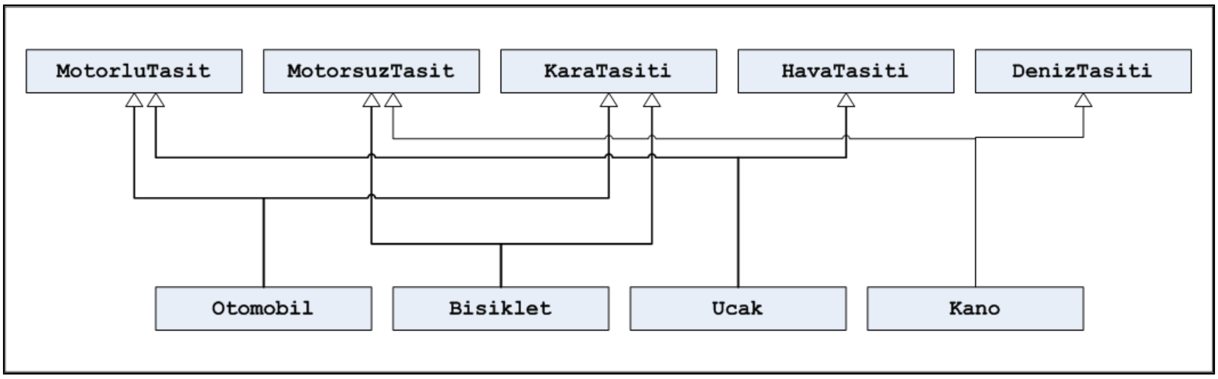
Arayüzler genelde kalıtım amacıyla değil, arayüz kavramı anlatılırken irdelenen, "sistemler arasında bütünleştirme noktaları oluşturma" amacıyla kullanılırlar. Eğer geliştirilen bir sistemin başka sistemlerle bütünleştirilebilmesi isteniyorsa, bu bütünleştirme tanımlanan arayüzler yardımı ile gerçekleştirilir.

Arayüzler ile Çoklu Kalıtım (*Multiple Inheritance*)

Java'da arayüzlerin bir başka kullanım alanı çoklu kalıtımdır. C++ gibi bazı diller bir sınıfın birden fazla sınıftan kalıtımla özelleştirilmesine olanak verirken, Java'da bir sınıfın **`extends`** anahtar sözcüğünden sonra birden fazla sınıfın adı yazılarak bu sınıflardan özelleştirilmesi olanaklı değildir.

Bununla birlikte, kalıtım çoğu zaman nitelikler için değil davranışlar (yöntemler) için yapılır. Bu gibi durumlarda özelleşen veri değil davranıştır

ve bu davranışlar ata sınıfta soyut tanımlanır. İşte bu noktada eğer soyut yöntemler için kalıtım yapılacaksa arayüzler devreye girebilir. Çünkü bir sınıf, gerçekleştirdiği arayüzdeki bütün yöntemleri içermekte ve birden fazla arayüzü gerçekleştirebilmektedir. Böylece, adı kalıtım olmasa da dolaylı yoldan çoklu kalıtım gerçekleştirilebilir. Aşağıda çoklu kalıtımın arayüzler ile nasıl gerçekleştirildiği görülmektedir.



Şekil 5-15. Çoklu kalıtım – Kalıtım ağacı

```
public interface MotorluTasit {
      // motorlu taşıtlarla ilgili yöntemler
}
```

Kod 5-109. Çoklu kalıtım - MotorluTasit.java

```
public interface MotorsuzTasit {
      // motorsuz taşıtlarla ilgili yöntemler
}
```

Kod 5-110. Çoklu kalıtım - MotorsuzTasit.java

```
public interface KaraTasiti {
      // kara taşıtlarıyla ilgili yöntemler
}
```

Kod 5-111. Çoklu kalıtım - KaraTasiti.java

```
public interface HavaTasiti {
      // hava taşıtlarıyla ilgili yöntemler
}
```

Kod 5-112. Çoklu kalıtım - HavaTasiti.java

```
public interface DenizTasiti {
      // deniz taşıtlarıyla ilgili yöntemler
}
```

Kod 5-113. Çoklu kalıtım - DenizTasiti.java

```
public class Otomobil implements MotorluTasit, KaraTasiti {
      // MotorluTasit ve KaraTasiti arayüzlerinden gelen yöntemlerin
      // gerçekleştirimleri
}
```

Kod 5-114. Çoklu kalıtım - Otomobil.java

```
public class Bisiklet implements MotorsuzTasit, KaraTasiti {
      // MotorsuzTasit ve KaraTasiti arayüzlerinden gelen yöntemlerin
      // gerçekleştirimleri
}
```

Kod 5-115. Çoklu kalıtım - Bisiklet.java

```
public class Ucak implements MotorluTasit, HavaTasiti {
      // MotorluTasit ve HavaTasiti arayüzlerinden gelen yöntemlerin
      // gerçekleştirimleri
}
```

Kod 5-116. Çoklu kalıtım - Ucak.java

```
public class Kano implements MotorsuzTasit, DenizTasiti {
      // MotorsuzTasit ve DenizTasiti arayüzlerinden gelen yöntemlerin
      // gerçekleştirimleri
}
```

Kod 5-117. Çoklu kalıtım - Kano.java

Bu örnekte de görüldüğü gibi, davranış kalıtımında arayüzler kullanılabilir ve bir sınıf birden fazla arayüzü gerçekleştirebilmesi sayesinde çoklu kalıtım sağlanabilir.

Benzer kalıtım ilişkileri soyut sınıflar ya da soyut olmayan sınıflar ve arayüzler bir arada kullanılarak farklı şekillerde oluşturulabilir. Burada tek doğru yoktur. Önemli olan, oluşturulan modelin doğru, tutarlı ve genişletilebilir olmasıdır. Gerek arayüzler, gerek soyut olan ya da olmayan sınıflar bu gereksinimleri karşılayacak şekilde dilendiği gibi kullanılabilir.

Arayüzler Arasındaki Kalıtım İlişkisi

Bir sınıf ile başka bir sınıf arasında kalıtım (*extends*) ilişkisinin kurulabileceğini biliyoruz. Bir sınıf ile bir ya da daha fazla arayüz arasında da gerçekleştirme (*implements*) ilişkisinin kurulabileceğini gördük.

Bir arayüz ile başka bir arayüz arasında ise gene kalıtım ilişkisi kurulabilir. Aynen sınıflar arasında olduğu gibi **`extends`** anahtar sözcüğü ile bir

arayüzün başka bir arayüzü özelleştirdiği belirtilebilir. Bu durumda, ata arayüzü özelleştiren alt arayüz, atasındaki yöntemleri alır.

Aşağıda iki arayüz arasında kalıtım ilişkisinin nasıl kurulduğu ve alt arayüzü gerçekleştiren sınıf görülmektedir:

```
public interface AtaArayuz {
      void islev1();
      void islev2();
}
```

Kod 5-118. Arayüzler arasında kalıtım – AtaArayuz.java

```
public interface AltArayuz extends AtaArayuz {
      void islev2();
      void islev3();
}
```

Kod 5-119. Arayüzler arasında kalıtım – AltArayuz.java

```
public class OrnekSinif implements AltArayuz {
      public void islev1() {
            // islev1 gerçekleştirimi olan kodlar
      }
      public void islev2() {
            // islev2 gerçekleştirimi olan kodlar
      }
      public void islev3() {
            // islev3 gerçekleştirimi olan kodlar
      }
}
```

Kod 5-120. Arayüzler arasında kalıtım – OrnekSinif.java

5.7 Aykırı Durumlar (*Exceptions*)

Bir tasarım yapılırken, uygulamanın çalışması sırasında ne olmaması gerektiğine değil, ne olması gerektiğine odaklanmak gerekir. Çünkü geliştirilen uygulamanın kullanılması sırasında genellikle kullanıcıların beklenen biçimde davranacakları, girdilerin beklendiği şekilde olacağı ve herhangi bir şekilde sorun oluşmayacağı varsayılır.

Uygulamalardan öncelikle gereksinimleri doğru bir şekilde karşılamaları beklenir. Çünkü bu gereksinimler, uygulamanın kullanıldığı sürenin çok büyük bir kesimini oluşturacaktır. Birçok kullanıcı çoğu zaman çok tekdüze ve basit işler yaparlar ancak bu işler hemen hemen bütün çalışma vakitlerini alır. Böyle bir kullanıcı için önemli olan, uygulamanın bir takım özel durumlarda oluşabilecek sorunlar karşısında nasıl davrandığı değil, en temel gereksinimleri ne kadar kolay ve doğru karşıladığıdır.

Hangi teknoloji, dil ya da programlama yaklaşımı ile geliştirildiğinden bağımsız olarak, çoğu uygulamanın kullanıcı profili bahsedildiği gibidir. Buna ek olarak, uygulamaların başarısını belirleyen de gene bu tip kullanıcılardır. Çünkü bu kullanıcılar çoğu zaman uygulamayı en yoğun kullanan ve iyi ya da kötü değerlendirmelerini en rahat dile getiren kişilerdir.

Dolayısıyla, bir uygulama tasarlanırken öncelikle uygulamanın temel gereksinimleri üzerinde durulur. Uygulamanın çalışması sırasında oluşabilecek bir takım özel durumlar (örneğin bir dosyanın bulunamaması ya da bağlanılacak başka bir makinanın o anda yanıt vermiyor olması.. gibi) yerine, bu özel durumların oluşmadığı senaryolar irdelenir. Uygulamanın tasarımında izlenen bu yol sayesinde, nadiren oluşabilecek özel durumlar yerine, her zaman kullanılacak temel gereksinimlere daha fazla emek harcanmış olur.

Benzer yaklaşım kodlama sırasında da kendisini gösterir. Eğer ne olması gerektiğinden çok ne olmaması gerektiğine odaklanılırsa, ortaya çıkacak kod birçok denetim deyimiyle karmaşıklaşmış, yönetilmesi ve

genişletilmesi zor bir kod haline gelecektir. Bu durumu şöyle bir senaryo ile örnekleyelim:

Örnek senaryo: Geliştireceğimiz program dosyalarla çalışıyor; kullanıcının girdiği bir takım bilgiler dosyalara kaydediliyor, dosyalardaki bilgiler kullanıcı tarafından incelenebiliyor ya da güncellenebiliyor olsun. Tipik bir işlem, belirli bir dosyadaki bilgileri okuma işlemi olacaktır. Şimdi bu tipik işlemi, önce oluşabilecek sorunları ele alarak, sonra da hiç sorun olmayacakmış gibi davranarak gerçekleştiren algoritmalara bakalım:

```
1- dosya adını al
2- bu isimde bir dosya varsa;
      2-1- dosyayı okuma amaçlı olarak aç
      2-2- dosya açılabildiyse;
            2-2-1- veri oku
            2-2-2- okumada sorun yoksa;
                  2-2-2-1- işlemi yap
                  2-2-2-2- dosyayı kapat
                  2-2-2-3- dosya kapatılamadıysa
                        2-2-2-3-1- kullanıcıya sorunu bildir
                  2-2-2-4- çık
            2-2-3- okumada sorun varsa;
                  2-2-3-1- kullanıcıya sorunu bildir
                  2-2-3-2- dosyayı kapat
                  2-2-3-3- dosya kapatılamadıysa
                        2-2-3-3-1- kullanıcıya sorunu bildir
                  2-2-3-4- çık
      2-3- dosya açılamadıysa;
            2-3-1- kullanıcıya sorunu bildir
3- bu isimde bir dosya yoksa;
      3-1- kullanıcıya sorunu bildir
4- çık
```

Kod 5-121. Dosyadan veri okuma, olası sorunlar ele alınırsa

Bu algoritmada “olası” sorunlu durumlar ele alınmıştır. Bu olası sorunlu durumların ele alınmasıyla birlikte ortaya çıkan sakıncalı durumlar ise şu şekilde özetlenebilir:

1- Görüldüğü gibi esas yapılmak istenen iş bir dosyadan veri okumaktır. Ancak bu iş ile ilgili kod kesimleri, olası sorunlu durumları ele alan kod kesimlerinin arasında kaybolmuş durumdadır. Gerçek işi yapan kod kesimi az olsa da o kod kesiminin bağlı olduğu koşulları içeren kod kesimleri oldukça kalabalıktır. Dolayısıyla kodun bakımı zorlaşmaktadır.

2- Kullanıcılar bu programı çalıştırırlarken dosyanın bulunamaması, açılamaması ya da dosyadan veri okunamaması olasılıkları oldukça düşüktür. 100 çalıştırmadan belki 99, belki 100 defasında sorun çıkmayacak olmasına rağmen, kodlamaya harcanan emek olası sorunlu durumlar için daha fazladır. Oysa kullanıcılar çoğunlukla programın sorunsuz yüzünü görecek ve değerlendireceklerdir.

3- Bütün olası sorunlu durumları ele almak olanaklı olmayabilir. Unutulan olası sorunlar da bulunabilir. Bu durumda harcanan emek de boşa gitmiş olur.

```
1- dosya adını al
2- dosyayı okuma amaçlı olarak aç
3- veri oku
4- işlemi yap
5- dosyayı kapat
6- çık
```

Kod 5-122. Dosyadan veri okuma, olası sorunlar ele alınmazsa

Bu algoritma ise hiç sorun çıkmayacakmışcasına işlem yapmaktadır. Bir önceki algoritma ile karşılaştırıldığında kodun çok daha temiz, kolay anlaşılır ve yönetilebilir olduğu görülmektedir. Ayrıca, oluşması beklenmeyen durumlar için emek de harcanmamıştır. Ancak küçük olasılıkla da olsa, eğer "olası" sorunlu durumlardan birisi gerçekleşirse ne yapılacağı belirlenmemiş olduğundan, kullanıcı kötü sürprizlerle karşılaşabilir ve bu da uygulamanın ve dolayısıyla uygulama geliştiricisinin güvenilirliğini zedeleyecektir.

Görüldüğü gibi her iki algoritmanın da bazı sakıncaları bulunmaktadır. İşte bu noktada Aykırı Durumlar'dan bahsetmek gerekir.

Aykırı durum, kodun çalışması sırasında karşılaşılması beklenmeyen ("*aykırı*") bir durum oluştuğu taktirde, kodu işleten kesimin (Java Sanal Makinası) devreye girerek programın akışını kestiği ve gerçekleşen aykırı durumu belirten bir nesne oluşturarak programa verdiği bir düzenektir. Nesnenin oluşturulup programa verilmesine **aykırı durum fırlatma** (*throwing exception*) adı verilmektedir.

Aykırı durumlar sayesinde, yukarıda bahsettiğimiz iki farklı algoritma ile ortaya çıkan sakıncalı durumlardan kurtulmak olanaklıdır. Aykırı durum düzeneği kullanılarak, yapılacak işin doğası gereği karşılaşılması beklenmeyen, o senaryo için gerçekten de aykırı olan durumların bir takım denetim deyimleri ile ele alınarak kodun karmaşıklaştırılması engellenebileceği gibi, ele alınmamış bir sorunun oluşması sonucunda programın kontrolsüz bir şekilde çakılması olasılığı da ortadan kaldırılabilir. Böylece hem daha kolay yönetilebilir bir kod ortaya çıkmış, hem de kullanıcının karşısına, programın çakıldığını gösteren anlamsız hata ekranları çıkmadan çeşitli önlemler alınmış olur.

Çeşitli dillerde aykırı durum düzenekleri bulunmaktadır ve bunlar genelde aynı mantıkla çalışmaktadır. Bu bölümde Java'nın aykırı durum düzeneği incelenecektir.

5.7.1 `try – catch` bloğu

Aykırı durum nesnesinin oluşturulup programa verilmesine aykırı durumun fırlatılması demiştik. Nesneyi oluşturan ve fırlatan kesim Java Sanal Makinesidir (kendi aykırı durum nesnelerimizi de fırlatabiliriz, ancak bu konu daha sonra açıklanacaktır). Bu nesne, beklenmeyen bir durumun oluştuğunu belirtir ve o durum ile ilgili çeşitli bilgiler içerir. Nesneyi oluşturmak için kullanılacak olan sınıflar da Java Platformunun bir parçası olan Java API'de tanımlanmıştır.

Fırlatılan bir aykırı durumun ele alınabilmesi için aykırı durumun oluşabileceği kod kesimi özel deyimler arasına yazılır. Bu deyimler **`try`** ve

catch deyimleridir. **try** ve **catch** deyimleri arasında kalan kod kesimine de **try - catch** bloğu adı verilir.

try – catch bloğunun yazılışı aşağıdaki gibidir:

```
deyim1;
try {
      deyim2;
      deyim3;
}
catch (Exception1 e) {
      deyim4;
}
catch (Exception2 e) {
      deyim5;
}
deyim6;
```

Kod 5-123. try – catch bloğunun yazılışı

try deyimini takip eden **{** ile **try** bloğu başlar. Bu kesime yazılan kodlar programlama dili açısından doğru olan her türlü kodu içerebilir. Bu kesim **}** ile kapatılır.

} işaretinin hemen arkasından bir **catch** deyimi gelmelidir. En az bir tane **catch** yazılmak zorundadır. Birden fazla **catch** deyimi de yazılabilir. **catch** deyimi, **try** bloğunun işletilmesi sırasında herhangi bir aykırı durum fırlatılırsa, fırlatılan aykırı duruma bağlı olarak işletilmesi istenen kodun yazılacağı yerdir. **try** bloğunda farklı aykırı durumların oluşma olasılığı varsa, hangi **catch** bloğunun işletileceğine **catch** deyimine yazılan sınıf bilgisine bakılarak karar verilir.

> ***catch deyimine ait olan ( ) işaretleri arasına, bir aykırı durum sınıfına ait bir referans değişken tanımı yapılmaktadır. Eğer fırlatılan aykırı durum bu catch deyimi tarafından yakalanırsa, referans değişken yakalanan aykırı durum nesnesinin referansı olacaktır. Böylece aykırı durum nesnesi ile ilgili bilgilere erişilebilecektir.***

Şimdi Kod 7-108'e dayanarak üreteceğimiz farklı senaryolarla **`try`** – **`catch`** bloğunun nasıl işletildiğine bakalım:

1. senaryo - Aykırı durum fırlatılmıyor: **`try`** bloğu işletilirken herhangi bir aykırı durum fırlatılmazsa, **`catch`** deyimlerine bakılmaz ve en son **`catch`** deyiminin bittiği yerden kod işletilmeye devam eder. Eğer herhangi bir aykırı durum oluşmazsa işletilecek deyimler aşağıdaki gibi olacaktır.

```
deyim1
deyim2
deyim3
deyim6
```

Çıktı 5-14. Aykırı durum fırlatılmadı

2. senaryo – **`deyim2`** *işletilirken* **`Exception1`** *türünden aykırı durum fırlatılıyor*: **`try`** bloğu işletilirken herhangi bir aykırı durum fırlatılırsa, **`catch`** bloklarından ilkine sapılır. Eğer **`catch`** deyimine yazılan sınıf adı, aykırı durum nesnesinin ait olduğu sınıf ile aynı türden ya da onun ata sınıfı ise, bu **`catch`** bloğunun içindeki kodlar işletilir. Eğer aykırı durum nesnesi ile **`catch`** deyimindeki sınıf adı arasında böyle bir ilişki yoksa, takip eden **`catch`** deyimine sapılarak aynı işlem tekrarlanır. **`catch`** bloklarından yalnızca bir tanesi işletilir. Herhangi bir **`catch`** bloğunun içindeki kodlar işletilmişse sonraki **`catch`** deyimlerine bakılmaz ve en son **`catch`** deyiminin bittiği yerden kod işletilmeye devam eder.

> ***Bir `catch` bloğu işletildikten sonra, akış `catch` bloklarının sonuncusunun bitişinden devam eder. Aykırı durum yakalandıktan sonra aykırı durumun fırlatıldığı satıra geri dönülmez!***

`deyim2` işletilirken **`Exception1`** sınıfına ait bir aykırı durum fırlatılırsa, işletilecek deyimler aşağıdaki gibi olacaktır.

```
deyim1
deyim4
deyim6
```

Çıktı 5-15. deyim2 işletilirken Exception1 aykırı durumu fırlatıldı

*3. senaryo – **deyim2** işletilirken **Exception2** türünden aykırı durum fırlatılıyor*: Bir önceki senaryoda verdiğimiz bilgilere dayanarak, **deyim2** işletilirken **Exception2** türünden bir aykırı durum fırlatılması durumunda işletilecek deyimlerin aşağıdaki gibi olacaklarını söyleyebiliriz:

```
deyim1
deyim5
deyim6
```

Çıktı 5-16. deyim2 işletilirken Exception2 aykırı durumu fırlatıldı

*4. senaryo – **deyim2** işletilirken **Exception2** sınıfının alt sınıfı olan **Exception2Alt** türünden aykırı durum fırlatılıyor*: 2. senaryoda belirttiğimiz gibi, bir **catch** deyimi ile aykırı durumun yakalanabilmesi için o aykırı durum nesnesinin ait olduğu sınıfın **catch** deyimindeki sınıf ile aynı ya da onun alt sınıfı olması gerekir. Bu senaryoda, fırlatılan aykırı durum **Exception2** sınıfının bir alt sınıfı olan **Exception2Alt** sınıfına ait olduğuna göre işletilecek deyimler aşağıdaki gibi olacaktır:

```
deyim1
deyim5
deyim6
```

Çıktı 5-17. deyim2 işletilirken Exception2Alt aykırı durumu fırlatıldı

> ***catch deyimine yazılan bir sınıf, kendi sınıfından ve alt sınıflarından olan bütün aykırı durum nesnelerini yakalayabilir. catch deyimlerinden yalnızca bir tanesi işletilebildiğine göre, bir catch deyiminin altına o catch deyiminde belirtilen sınıfın alt sınıflarından herhangi birisi ile başka bir catch deyimi yazmak anlamsız olacaktır. Zaten derleyici de bu durumda derleme hatası verir.***

*5. senaryo – **deyim2** işletilirken **Exception3** türünden aykırı durum fırlatılıyor*: **catch** bloklarından hiçbirisinde **Exception3** türüne ait bir bildirimde bulunulmadığı ve **Exception1** ya da **Exception2** sınıfları **Exception3** sınıfının ata sınıfı olmadığı için bu kod örneğinde aykırı durum yakalanamaz. Bir aykırı durum fırlatılmış ve **catch** deyimlerinden

hiçbirisinde o aykırı durum yakalanamamışsa ne olur? Kodun akışı nasıl devam eder?

Eğer henüz yakalanmamış bir aykırı durum varsa, o aykırı durum yakalanana kadar hiçbir kod işletilmez (**finally** bloğu bu genellemenin dışındadır, ilerleyen kesimde anlatılacaktır). Aykırı durumun fırlatıldığı kod kesimi bir **try – catch** bloğunun içindeyse, o **catch** bloklarında uygun (**catch** deyimindeki sınıfın aykırı durum ile aynı sınıftan ya da onun ata sınıfı olduğu) bir **catch** deyimi aranır. Bulunamazsa, bir üst kod bloğuna çıkılarak, o kod bloğuna ait bir **catch** deyimi aranır. Örneğin eğer bir yöntem işletilirken aykırı durum oluşmuşsa, bu yöntemin çağrıldığı kod kesiminin içinde bulunduğu **try – catch** bloğuna ait olan **catch** deyimlerine bakılır. Bu akış, aykırı durum yakalanana kadar devam eder. Eğer aykırı durum herhangi bir **catch** bloğunda yakalanırsa, o **catch** bloğundan sonraki **catch** bloklarının en sonuncusunun bittiği yerden kod işletilmeye devam eder. Aykırı durum yakalanamazsa, bir üst kod bloğuna çıkışlar programın başladığı **main** işlevine kadar sürer. Burada da aykırı durumu yakalayacak deyimler bulunamazsa, programın çalışması sona erer. Aykırı durum, programı çalıştıran Java Sanal Makinesi tarafından yakalanır ve konsola aykırı durum ile ilgili bir takım bilgiler (aykırı durumun ne olduğu, açıklaması, hangi satırda oluştuğu ve o satıra gelinene kadar hangi işlevlerden geçildiği) yazılır.

6. senaryo – **deyim2** *işletilirken* **Exception1** *türünden aykırı durum oluşuyor. Sonra* **deyim4** *işletilirken* **Exception2** *türünden bir aykırı durum oluşuyor*: Bu senaryoda oluşan aykırı durum ilk **catch** deyimi ile yakalanmış ancak **catch** bloğu işletilirken yeni bir aykırı durum oluşmuştur. İlk bakışta, "oluşan aykırı durum **Exception2** sınıfına ait olduğuna göre hemen bir sonraki **catch** deyimi ile yakalanır ve **deyim5** işletilir" diye düşünülebilir. Ancak daha önce verdiğimiz bilgilere göre, aynı **try** bloğuna ait **catch** bloklarından yalnızca bir tanesi işletilebilir. Bu durumda, **deyim4** işletilirken oluşacak aykırı durum, bir sonraki **catch** deyimi tarafından

yakalanamaz. 5. senaryoda açıkladığımız gibi, bu aykırı durumun yakalanabilmesi için bir üst kod bloğuna bakılacaktır. Buradan itibaren akış 5. senaryodaki gibidir.

5.7.2 `finally` deyimi

Bir **`try – catch`** bloğunun sonuna **`finally`** deyimi yazılabilir. **`finally`** deyiminin yazılması zorunlu değildir. Eğer yazılacaksa, en son **`catch`** bloğunu takip etmelidir.

`finally` bloğu, aykırı durum oluşsun oluşmasın, yakalansın yakalanmasın her durumda işletilmesi istenen kod kesimlerinin yazıldığı yerdir. Bu ifadeyi yine bir kod örneği ve yukarıdaki senaryolar ile açıklamaya çalışalım.

```
deyim1;
try {
	deyim2;
	deyim3;
}
catch (Exception1 e) {
	deyim4;
}
catch (Exception2 e) {
	deyim5;
}
finally {
	deyim6;
}
deyim7;
```

Kod 5-124. try – catch – finally yazılışı

1. senaryo - Aykırı durum fırlatılmıyor: **`try`** bloğu işletilirken herhangi bir aykırı durum fırlatılmazsa, **`catch`** deyimlerine bakılmaz. **`finally`** bloğu işletilir ve **`finally`** bloğunun bittiği yerden kod işletilmeye devam eder:

```
deyim1
deyim2
deyim3
```

```
deyim6
deyim7
```

Çıktı 5-18. Aykırı durum fırlatılmadı

2. senaryo – **`deyim2`** *işletilirken* **`Exception1`** *türünden aykırı durum fırlatılıyor*: Aykırı durum ilk **`catch`** deyimi ile yakalandıktan sonra **`finally`** bloğu işletilir ve **`finally`** bloğunun bittiği yerden kod işletilmeye devam eder:

```
deyim1
deyim5
deyim6
deyim7
```

Çıktı 5-19. deyim2 işletilirken Exception1 aykırı durumu fırlatıldı

3. senaryo – **`deyim2`** *işletilirken* **`Exception2`** *türünden aykırı durum fırlatılıyor*: Aykırı durum ikinci **`catch`** deyimi ile yakalandıktan sonra **`finally`** bloğu işletilir ve **`finally`** bloğunun bittiği yerden kod işletilmeye devam eder:

```
deyim1
deyim4
deyim6
deyim7
```

Çıktı 5-20. deyim2 işletilirken Exception2 aykırı durumu fırlatıldı

4. senaryo – **`deyim2`** *işletilirken* **`Exception2`** *sınıfının alt sınıfı olan* **`Exception2Alt`** *türünden aykırı durum fırlatılıyor*: Aykırı durum ikinci **`catch`** deyimi ile yakalandıktan sonra **`finally`** bloğu işletilir ve **`finally`** bloğunun bittiği yerden kod işletilmeye devam eder:

```
deyim1
deyim4
deyim6
deyim7
```

Çıktı 5-21. deyim2 işletilirken Exception2Alt aykırı durumu fırlatıldı

5. senaryo – **`deyim2`** *işletilirken* **`Exception3`** *türünden aykırı durum fırlatılıyor*: **`catch`** bloklarından hiçbirisinde **`Exception3`** türüne ait bir

bildirim bulunmadığına göre bu aykırı durum yakalanamaz. **`finally`** bloğu işletilir ve bir üst kod bloğuna sapılır:

```
deyim1
deyim6
```

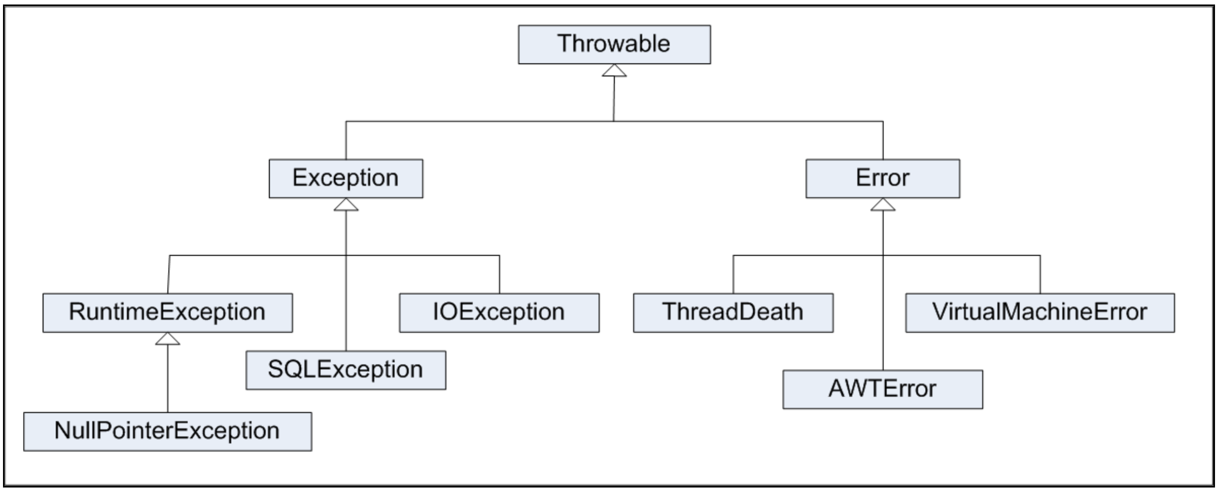
Çıktı 5-22. deyim2 işletilirken Exception3 aykırı durumu fırlatıldı

*6. senaryo – **`deyim2`** işletilirken **`Exception1`** türünden aykırı durum oluşuyor. Sonra **`deyim4`** işletilirken **`Exception2`** türünden bir aykırı durum oluşuyor*: Bu senaryoda oluşan aykırı durum ilk **`catch`** deyimi ile yakalanmış ancak **`catch`** bloğu işletilirken yeni bir aykırı durum oluşmuştur. **`finally`** bloğu işletildikten sonra kodun akışı bir üst kod bloğuna sapacaktır.

5.7.3 Java'da Aykırı Durumlar

Aykırı durum düzeneğinin çalışmasını örneklerle pekiştirmeden önce Java API ile gelen aykırı durum sıradüzenini incelemekte fayda var.

Aşağıdaki şekil Java'daki aykırı durum sıradüzenini (*hierarchy*) özetlemektedir. Burada, var olan aykırı durumların çok küçük bir alt kümesi görülmektedir. Amaç bütün aykırı durumları tanıtmak ya da anlatmak değil, birazdan açıklayacağımız aykırı durum sınıflandırmasının nereden geldiğini göstermektir. Aykırı durumlarla ilgili daha geniş bilgi almak için Java API Belirtimi belgelerini incelemek gerekir.



Şekil 5-16. Java'da aykırı durum sıradüzeni

`Throwable` sınıfı bütün aykırı durumların atasıdır. Bir aykırı durum nesnesinin "fırlatılabilmesi" için, o nesnenin ait olduğu sınıfın `Throwable` sınıfının doğrudan ya da dolaylı alt sınıfı olması gerekir. `Throwable` sınıfının iki tane doğrudan alt sınıfı bulunmaktadır: `Exception` ve `Error` sınıfları.

`Error` sınıfı ve onun alt sınıfları, bir uygulamanın ele alması beklenmeyen, anormal koşullar sonucunda oluşmuş hataları belirtirler. Bunlar normal koşullarda meydana gelmemesi gereken durumlardır.

`Exception` sınıfı ve onun alt sınıfları ise uygulamanın kodunda `try – catch` blokları ile ele alınabilecek aykırı durumların oluşturulduğu sınıflardır.

Denetlenen Aykırı Durumlar (*Checked Exceptions*)

Şekil 7-15'te görülen aykırı durum sıradüzeninde, sol tarafta bulunan ve `Exception` sınıfı ile başlayan kalıtım ağacında `RuntimeException` sınıfı ile başlayan alt ağacın dışında kalan bütün aykırı durumlar denetlenen aykırı durumlardır. Örneğin Şekil 7-15'teki `Exception`, `IOException` ve `SQLException` sınıfları denetlenen aykırı durum sınıflarıdır.

Soru: "denetlenen aykırı durum" ne demektir? Denetleme işini kim yapmaktadır?

Eğer işletilecek bir kod kesiminde, denetlenen aykırı durum olarak adlandırılan sınıflardan herhangi birisine ait bir aykırı durum oluşması olasılığı varsa, programcının bu aykırı durum oluşma olasılığının farkında olduğunu derleyiciye bildirmesi gerekir. Başka bir deyişle **"denetlenen aykırı durumlar", ele alınıp alınmadıkları derleyici tarafından denetlenen aykırı durumlardır**. Eğer bu aykırı durumlar ele alınmamış ise derleme hatası oluşur.

Programcının bu bildirimi yapabilmesinin iki yolu bulunmaktadır.

Bunlardan ilki, denetlenen aykırı durumun fırlatılma olasılığı bulunan kod kesimini bir `try – catch` bloğuna yazmak ve `catch` deyimlerinden bir tanesinde bu aykırı durumu ele alabilecek bildirimin yapılmasıdır.

İkinci yol ise **throws** deyimini kullanmaktır. Eğer bir kod kesiminde denetlenen bir aykırı durum fırlatılma olasılığı varsa ve programcı **try – catch** bloğunu yazmak ya da yazdığı **try – catch** bloğunda bu aykırı durumu ele almak istemiyorsa, kod kesiminin içinde bulunduğu yöntemin bildiriminin sonuna **throws** deyimi yazılır.

Örneğin aşağıdaki kod örneğinde **deyim1** işletilirken **Exception1** adlı bir denetlenen aykırı durumun fırlatılması olasılığı, yöntem bildiriminin sonundaki **throws** deyimi ile belirtilmiştir.

```
public void islev() throws Exception1 {
      ....
      deyim1;
      ....
}
```

Kod 5-125. throws deyimi ile bir aykırı durumun fırlatılabileceğinin belirtilmesi

Soru: Oluşabilecek bir aykırı durum niçin **try – catch** bloğu ile ele alınmaz da **throws** deyimi ile yöntemin o aykırı durumu fırlatabileceği belirtilir?

Bir yöntem, aldığı parametreler ile çalışan bir kod kesimidir. Yöntemin içindeki kod kesimi, programın geri kalan kısımlarında neler olup bittiği ile ilgilenmez/ilgilenmemelidir. Bir yöntemin (genel olarak her işlevin) yalnızca kendisine verilen parametrelere dayanarak tanımlı bir işi (yalnızca 1 işi) yapması, kodun modülerliğini, değiştirilebilirliğini arttıran ve bakımını kolaylaştıran bir programlama tekniğidir. Bazı durumlarda, yöntem çalışırken bir aykırı durum oluşursa ne yapılması gerektiği o yöntemi çağıran kod kesiminde karar verilebilecek bir konudur. Dolayısıyla, yöntemden tek beklenen, yapmak üzere çağrıldığı tanımlı işi başarıyla gerçekleştirmesi, eğer gerçekleştiremiyorsa sorunun ne olduğunu çağrıldığı yere bildirmesidir.

Denetlenen aykırı durumlar, derleyici tarafından ele alınmaları zorlanan aykırı durumlardır. Eğer yöntemin için denetlenen aykırı durum sınıflarından herhangi birisine ait bir aykırı durumun fırlatılması olasılığı

varsa, bu aykırı durumun ele alınması gerekecektir. Ancak az önce açıkladığımız nedenlerle bu aykırı durumun **`try – catch`** ile ele alınması doğru olmayabilir. İşte bu durumda **`throws`** deyimi devreye girer.

Denetlenmeyen Aykırı Durumlar (*Unchecked Exceptions*)

Denetlenen aykırı durum sınıflarının dışında kalan ve **`Exception`** sınıfının alt sınıfı olan bütün sınıflar (yani **`RuntimeException`** sınıfı ve onun doğrudan ya da dolaylı bütün alt sınıfları) denetlenmeyen aykırı durumlardır. Denetlenmeyen aykırı durumlar, ele alınmaları derleyici tarafından zorlanmayan aykırı durumlardır ve bu tip aykırı durumların ele alınıp alınmamaları programcının kararına bağlıdır. **`try – catch`** bloğu ile yakalanmalarında herhangi bir sakınca yoktur. Örneğin **`NullPointerException`** denetlenmeyen bir aykırı durumdur ve değeri **`null`** olan (herhangi bir nesneyi göstermeyen, nesne adresi tutmayan) bir referans değişken kullanılarak bir yöntem ya da niteliğe erişilmeye çalışılırsa oluşur.

5.7.4 Programcının Kodladığı Aykırı Durumlar

Programcılar isterlerse kendi aykırı durum sınıflarını da tanımlayabilirler. Bir aykırı durum sınıfı tanımlamak için **`Throwable`** sınıfının doğrudan ya da dolaylı olarak alt sınıfı olan bir sınıf kodlamaktır. Ancak genelde **`Throwable`** sınıfının doğrudan alt sınıfını yazmak yerine, onun alt sınıfı olan **`Exception`** sınıfının doğrudan ya da dolaylı alt sınıfını yazmak tercih edilir.

Programcı, yazdığı aykırı durumun denetlenmeyen aykırı durum olmasını isterse **`RuntimeException`** sınıfı ile başlayan sıradüzen ağacının altında kalacak bir sınıf yazar. Bu ağacın dışında kalan bir aykırı durum ise denetlenen aykırı durum olacaktır.

> ***Aykırı durum sınıflarının isimleri genelde `Exception` sözcüğü ile sonlandırılır. Böylece o sınıfın bir aykırı durum sınıfı olduğu sınıfı adı ile belirtilmiş olur:***
>
> ***AcikOturumYokException.java, BaglantiSaglanamadiException.java***

Aşağıda biri denetlenen biri de denetlenmeyen aykırı durum olmak üzere iki adet aykırı durum sınıfı görülmektedir.

```
public class AcikOturumYokException extends Exception {
	public AcikOturumYokException() {
		super();
	}
	public AcikOturumYokException(String msg) {
		super(msg);
	}
	public AcikOturumYokException(Exception e) {
		super(e);
	}
}
```

Kod 5-126. AcikOturumYokException.java

```
public class BaglantiSaglanamadiException extends RuntimeException {
	public BaglantiSaglanamadiException() {
		super();
	}
	public BaglantiSaglanamadiException(String msg) {
		super(msg);
	}
	public BaglantiSaglanamadiException(Exception e) {
		super(e);
	}
}
```

Kod 5-127. BaglantiSaglanamadiException.java

Programcı tarafından yazılan aykırı durumlar, bazı koşulların oluşmasına bağlı olarak gene programcı tarafından fırlatılmak zorundadır. Örneğin yukarıdaki aykırı durumlardan AcikOturumYokException, uygulamayı kullanmak isteyen bir kullanıcının önce kullanıcı adı ve parola girerek sistemde bir oturum açmadığı durumda fırlatılabilir. Kullanıcının henüz oturum açıp açmadığını denetleyen kod kesimi, eğer oturum açılmamışsa bu aykırı durumu fırlatabilir.

İster programcı tarafından yazılmış bir aykırı durum sınıfı olsun, ister Java API ile gelen bir aykırı durum sınıfı olsun, bir aykırı durumun

fırlatılabilmesi için **`throw`** deyimi kullanılır. **`throw`** deyimi, kendisinden sonra gelen aykırı durum nesnesini fırlatır. Aşağıda nasıl kullanıldığı görülmektedir:

```
public static A islev(HttpSession session)
                throws AcikOturumYokException {

      if (session.getAttribute("oturumBilgileri") == null) {
            throw new AcikOturumYokException();
      }
      return new A();
}
```

Kod 5-128. throw deyiminin kullanılması

Buradaki örnekte bir if deyimi ile bir koşul denetlenmekte, koşulun doğru olması durumunda AcikOturumYokException sınıfına ait bir aykırı durum fırlatılmaktadır. AcikOturumYokException aykırı durumu RuntimeException sınıfının alt sınıfı olmadığından, denetlenen aykırı durumdur. O nedenle yöntemin tanımında throws deyimi ile bildirilmiştir.

> ***`throws` deyimi ile `throw` deyimini birbirine karıştırmamak gerekir. `throws` deyimi bir yöntemin içinde fırlatılabilecek bir aykırı durumu yöntemi çağıran kod kesimine bildirmekte, `throw` deyimi ise bir aykırı durum nesnesini fırlatmaktadır.***

Programcı Neden Kendi Aykırı Durum Sınfılarını Yazar?

Java Platformu zengin bir aykırı durum ailesi sunmaktadır. Bir uygulama geliştirilirken yeni aykırı durum sınıflarının tanımlanması yerine, var olan aykırı durum sınıflarının kullanılması uygulanabilir bir yöntemdir. Ancak bazı durumlarda programcılar kendi aykırı durum sınıflarını yazma gereği duyarlar.

Uygulama geliştirme çatılarını (*framework*) kodlayan kişi ya da gruplar, geliştirdikleri çatı ile ilgili aykırı durum sınıflarını tanımlarlar. Bu bağlamda Java API de bir uygulama geliştirme çatısı olarak görülebilir ve Java API ile gelen aykırı durumlar aslında bu bakış açısı ile değerlendirilmelidir. Java dünyasında bir çok uygulama geliştirme çatısı bulunmaktadır. Bunlar Web

tabanlı uygulama, veri tabanı uygulaması ya da dağıtılmış uygulamalar geliştirmek gibi amaçlar için oluşturulmuş çatılar olabilir (Struts, Spring, Hibernate, EJB...). Her uygulama çatısı kendi aykırı durum ailesi ile birlikte gelmektedir.

Programcıların kendi aykırı durumlarını yazmak istemelerinin bir başka nedeni de kodun okunurluğunu arttırmak ve yönetim kolaylığı getirmektir. Çünkü bir aykırı durum sınıfına verilen isim, oluşan özel durumu kolaylıkla ifade edebilir. `Exception` adı ile yakalanan bir aykırı durum nesnesine göre, `AcikOturumYokException` adı ile yakalanan bir aykırı durum çok daha açıklayıcıdır ve kodun bakımı açısından önemli olabilir. Dolayısıyla zaman zaman programcılar kendi aykırı durum sınıflarını tanımlama yolunu seçmektedirler.

5.7.5 Aykırı Durumların Yönetilmesi

Aykırı durum düzeneğinin öğrenilmesi kolaydır. Denetlenen aykırı durumlar da zaten derleyici tarafından tespit edileceklerdir. Dolayısıyla aykırı durumları kodlamakla ilgili ciddi bir zorluk yoktur.

Ancak esas önemli olan konu aykırı durumlar yakalandığında ne yapılacağıdır. Bu noktada aykırı durum yönetme stratejileri devreye girer.

Uygulamanın yapısına göre çeşitli farklı stratejiler belirlenebilir. Ancak bir takım önemli noktaları akılda tutmakta fayda vardır.

1. Nokta: Yalnızca gerçekten aykırı olan (beklenmeyen) durumları aykırı durum olarak kodlamak gerekir. Aykırı durum nesnesinin oluşturulmasının işlemci ve bellek açısından bir maliyeti vardır. Dolayısıyla, **oluşması beklenen** durumların aykırı durum olarak ele alınmaları yanlış olabilir. Örneğin bir bölme işleminde bölenin 0 (sıfır) olması olasılığı varsa, bu durum `DivisionByZeroException` aykırı durumu ile değil, basit bir `if` deyimi ile denetlenmelidir. Çünkü bölenin sıfır değerini alması zaten bekleniyordur ve bu durumun `if` deyimi ile denetlenmesi daha az maliyetlidir.

2. Nokta: Kendi aykırı durum sınıflarımızı oluşturmadan önce Java API ile gelen aykırı durum sıradüzenini incelemek gerekir. Oldukça zengin bir aykırı durum ağacı varken, bunları kullanmayıp kendi aykırı durumlarımızı yazmaya karar verirken iki kere düşünmekte fayda vardır.

3. Nokta: Her aykırı durumun tek tek ele alınması anlamlı olmayabilir. Bir `try` bloğu işletilirken fırlatılma olasılığı olan aykırı durumların herbirisi için ayrı bir `catch` deyimi yazmak, yazılması ve yönetilmesi gereken kod miktarını arttıracaktır. Eğer gerçekten de her aykırı durumun ayrı ayrı ele alınması gerekmiyorsa ya da aykırı durumun türüne göre yapılacak işlem değişmiyorsa bu yaklaşım sakıncalı olabilir. Çünkü yazılacak her satır kod bakımı yapılacak kod miktarını arttırmaktadır. Birden fazla aykırı durum ortak bir ata sınıf ile ele alınabilir.

4. Nokta: Bütün aykırı durumların tek bir ata sınıf ile ele alınması anlamlı olmayabilir. `Exception` sınıfı bütün aykırı durumların atası olduğuna göre, herhangi bir `try` bloğunun sonuna `Exception` sınıfını (ya da fırlatılabilecek aykırı durumların ortak atası olan başka bir sınıfı) kullanarak tek bir `catch` deyimi yazmak ve bununla bütün aykırı durumları yakalamak olanaklıdır. Ancak bu yaklaşım da hangi aykırı durumun oluştuğu bilgisinin gerekli olduğu durumlarda sakıncalı olabilir.

5. Nokta: Bir aykırı durum yakalandığında yapılabilecek en kötü şey hiçbirşey yapmamaktır. Özellikle aykırı durum konusunu yeni öğrenen kişiler, bazen sırf derleyici zorladığı için `try – catch` yazarak denetlenen aykırı durumlara verilen derleme hatalarından kurtulma yolunu seçerler. Ancak genellikle aykırı durum oluştuğunda yapılacak birşey de bulunamaz ve `catch` bloğu boş bırakılır. Bu yaklaşım, yapılabilecek en kötü şeydir. Çünkü bir aykırı durumun oluşması yolunda gitmeyen birşeyler olduğunu ifade eder. Aykırı durum yakalanarak hiçbirşey yapmadan, herşey yolundaymış gibi programın çalışmaya devam etmesi kullanıcı için aldatıcı olabilir. Daha kötüsü, sorunlu durumun farkedilememesi nedeniyle uygulamanın düzeltilmesi olanağının da önü tıkanmış olur. Bir aykırı

durum yakalandığında yapılabilecek en basit işlem bu aykırı durumun oluştuğu bilgisini bir yerlere kaydetmektir (örneğin günlük dosyalarına).

Bir aykırı durum ile ilgili en temel ve açıklayıcı bilgilerin alınması için kullanılabilecek iki yöntem bulunmaktadır: **`getMessage()`** ve **`printStackTrace()`** yöntemleri.

Aykırı durumlar oluşturulurken, oluşan aykırı durumu açıklayan özet bilgi bir ileti olarak aykırı durum nesnesinin içine koyulur. **`getMessage()`** yöntemi, aykırı durum nesnesinin içindeki bu iletiyi döndürür. Oluşan aykırı durum ile ilgili temel bilgi verilmek istenirse bu ileti kullanılabilir.

Her yöntem çağrısı, çağrılan yönteme ait bazı bilgilerin (yöntemin yerel değişkenleri, parametreleri, dönüş değerleri ve adresleri) yığıt belleğe yazılmasına neden olur. Yığıttaki herhangi bir yöntem, hemen altındaki yöntem tarafından çağrılmıştır. **`printStackTrace()`** yöntemi, çağrıldığı anda yığıtta neler olduğunu ekrana döken yöntemdir. Böylece fırlatılan aykırı durumun hangi satırda, hangi işlevde oluştuğu ve o işlevin çağrılmasına kadar hangi işlevlerin arka arkaya çağrıldığı görülmüş olur. Fırlatılan bir aykırı durumun program içinde yakalan(a)maması durumunda bu aykırı durumun Java Sanal Makinesi tarafından yakalandığını ve ekrana bazı bilgilerin yazıldığını daha önce belirtmiştik. İşte bu bilgiler aslında Java Sanal Makinası tarafından **`printStackTrace()`** yöntemi çağrılarak yığıtın durumunun ekrana yazdırılmasından başka birşey değildir.

Bu yöntemler aykırı durum nesnesini gösteren referans değişken kullanılarak çağrılabilir. Aşağıda yöntemlerin kullanılışını örnekleyen kod görülmektedir.

```
try {
      deyim;
}
catch (Exception e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
}
```

Kod 5-129. getMessage() ve printStackTrace() yöntemlerinin kullanılışı

Görüldüğü gibi her iki yönteme de, yakalanan aykırı durum nesnesini gösteren değişken olan **`e`** referansı üzerinden ulaşılmaktadır.

6. Nokta: Bir aykırı durum yakalandığında ne yapılabileceğinin iyi değerlendirilmesi gerekir. Aykırı durum düzeneği programcıya önlem alma ve/ya da sorunun üstesinden gelme olanağı verir. Aykırı durumun fırlatılmasına neden olan koşulların değiştirilmesi ve başka bir senaryonun devreye sokulması olanaklı olabilir. Ancak bazı aykırı durumlar, çalışma zamanında üstesinden gelinemeyecek fiziksel sorunlardan kaynaklanır. Örneğin okunması gereken bir dosyanın bozulmuş olması (**`IOException`**) ya da bağlanılacak başka bir bilgisayara çeşitli nedenlerden bağlanılamaması (**`SocketException`**) gibi durumlar çalışma zamanında düzeltilebilecek sorunlar değildir. Genelde bu gibi koşullar oluştuğunda alınabilecek bir önlem ya da devreye sokulacak ikinci bir senaryo bulunmaz. Bununla birlikte bu gibi aykırı durumların çoğu denetlenen aykırı durumlar olarak tanımlanmıştır. Yani ya **`try – catch`** bloğu ile ya da **`throws`** deyimi ile ele alınmaları gerekmektedir.

`throws` deyiminin kullanılma amacı, denetlenen aykırı durumlarla ilgili derleme hatalarından kurtulmak değil, yöntemin aykırı durumla karşılaştığında ne yapacağını bilmediğini belirtmektir. Böylece o yöntemi çağıran kesimde aykırı durumun ele alınması beklenir. Eğer **`throws`** deyimi sırf denetlenen aykırı durumla ilgili derleme hatası almamak için kullanılırsa, birbirini çağıran yöntem tanımlarında zincirleme bir şekilde **`throws`** deyimi yazmak gerekir ve programın çalışmaya başladığı **`main`** yöntemine kadar yansıyacaktır. Kodun her yerine throws deyimi yazmak programlama açısından can sıkıcı bir durumdur ve kodu kirletmektedir. Bunun yerine bir noktada **`try – catch`** bloğu ile aykırı durumun ele alınması gerekir.

Kısaca söylersek, denetlenen aykırı durumlar genellikle **`try – catch`** bloğunun yazılmasını ve aykırı durumun yakalanmasını zorlar. Ancak yukarıda da bahsettiğimiz gibi bazı aykırı durumların çalışma zamanında çözülmeleri olanaklı değildir. Bunlar ya kodda değişiklik yapmayı ya da

bilgisayarı açıp kapatmak, ağ kablosunu takmak, yeni bir dosya kopyalamak gibi fiziksel müdahaleleri gerektirir.

Soru: Bazı denetlenen aykırı durumları **`try – catch`** bloğu ile yakalamak zorunda kalıyoruz ancak çalışma zamanında bu aykırı durumların üstesinden gelme şansımız yok. Ne yapmalıyız?

Bir aykırı durum yakalandığında yapılabilecek en kötü şeyin hiçbirşey yapmamak olduğunu, en azından bu aykırı durum ile ilgili bilgilerin bir günlük dosyasına ya da konsola yazılması gerektiğini söylemiştik. Şimdi bir adım daha ileri giderek, eğer çalışma zamanında üstesinden gelinemeyecek bir aykırı durum ile karşılaşılmışsa programın denetimli bir şekilde sonlandırılabileceğini ifade edelim. Kullanıcının karşısına anlamlı bir ekran ya da ileti çıkartarak bir sorun olduğunu, bu sorunun nasıl çözülebileceğini anlattıktan sonra programın sonlandırılması anlamlıdır. Böylece kullanıcı o sorun ile ilgili önlem aldıktan sonra programı yeniden çalıştırabilir.

Bunu yapabilmenin en kestirme yolu, denetlenen aykırı durumun yakalandığı noktada bu aykırı durum günlük dosyasına yazıldıktan hemen sonra yeni bir aykırı durum fırlatmak ve bu aykırı durumu diğer benzer aykırı durumları da yakalayacak ve kullanıcının karşısına anlamlı bir uyarı ekranı çıkartacak daha üstlerdeki bir kod bloğunda ele almaktır. Tabi ki yeniden **`try – catch`** ya da **`throws`** deyimleri ile uğraşmamak için, fırlatılan yeni aykırı durumun denetlenmeyen bir aykırı durum olması gerekir. Aşağıda bununla ilgili küçük bir kod örneği görülmektedir.

```
try {
      deyim;
}
catch (SocketException e) {
      e.printStackTrace();
      throw new RuntimeException(e);
}
```

Kod 5-130. Denetlenen aykırı durumun yakalanarak denetlenmeyen aykırı durum fırlatılması

7. Nokta: Bazı aykırı durumlar programcı hatası nedeniyle oluşurlar. Bu aykırı durumların yakalanmayıp uygulamanın sonlanması anlamlı olabilir. Özellikle geliştirme sürecinde, bu tip aykırı durumların yakalanmaması koddaki hataların belirlenmelerine ve dolayısıyla düzeltilebilmelerine olanak sağlayacaktır. Örneğin **`NullPointerException`** aykırı durumu çoğu zaman programcı hatasının sonucudur. Eğer bir referans değişkenin değerinin **`null`** olması olağan bir durumsa, bu koşul basit bir **`if`** deyimi ile denetlenerek işlem yapılır. Ancak bunun dışında **`NullPointerException`** aykırı durumu fırlatılıyorsa bu büyük olasılıkla programcının hatasıdır. **`NullPointerException`** aykırı durumu denetlenmeyen bir aykırı durum olduğuna göre ele alınması derleyici tarafından zorlanmayacaktır. Dolayısıyla aykırı durum yakalanmayacak ve uygulama sonlanacak, Java Sanal Makinası aykırı durumun hangi satırda fırlatıldığını konsola yazacaktır. Böylece hatanın bulunup düzeltilmesi olanaklı olur.

5.8 Girdi/Çıktı (*Input/Output – I/O*) İşlemleri

Java platformu, girdi/çıktı işlemleri ile ilgili olarak farklı paketler altında toplanmış olan çok sayıda sınıf ve arayüz içermektedir. Farklı paketler, girdi/çıktı işlemleri için farklı soyutlama yaklaşımları sunmaktadır.

`java.io` paketi, "akış (*stream*)" olarak adlandırılan yapıların soyutlamaları ile ilgili sınıf ve arayüzleri içermektedir. Bunlar da kendi aralarında *byte*-tabanlı ve karakter-tabanlı olmak üzere ayrılırlar.

`java.nio` paketi, daha hızlı girdi/çıktı işlemleri gerçekleştirilebilmesi amacı ile tasarlanmış, "kanal (*channel*)" ve "yastık alan (*buffer*)" olarak adlandırılan soyutlamalar ve onlarla ilgili sınıf ve arayüzleri içerir.

`java.net` paketi, ağ tabanlı girdi/çıktı işlemleri için "soketler (*sockets*)" üzerinde odaklanmış, altta akış ya da kanal tabanlı modeli kullanan yapıları içerir.

Java'da girdi/çıktı işlemlerinin bu kadar çeşitlilik göstermesi, hem gereksinimlerdeki çeşitliliğin hem de girdi/çıktı kütüphanesinin geliştirilmesi sürecinin sonucudur. Dolayısıyla bu bölümde hem bu gereksinim çeşitliliğini hem de tarihsel sırada Java girdi/çıktı kütüphanelerinin gelişimini inceleyen bir bakış açısı ile ilerleyeceğiz. Bu sırada, özellikle programlamayı yeni öğrenen kişiler için zaman zaman kafa karıştırıcı olabilen bazı temel kavramlara değinmeyi de ihmal etmeyeceğiz.

Girdi/Çıktı işlemleri denilince ilk akla gelen genelde Dosya İşlemleri olmaktadır. Oysa Ağ İşlemleri de girdi/çıktı bağlamında ele alınmalıdır. Ancak bu kitapta hedeflediğimiz kitle ve belirlediğimiz düzey bakımından, bu kesimde yalnızca Dosya İşlemlerini ele alacağız.

Text File – Binary File

Writer - Stream

java.io.File

java.io.RandomAccessFile

io package

nio package

serialization – transient - volatile

decorator pattern

5.9 Derlemler (*Collections*)

.

5.10 JavaBean Kavramı

Java sınıfları ile ilgili olarak sarmalama ilkesi ve erişim yöntemlerini anlattıktan hemen sonra, Java dünyasında çok anılan ve önemli bir yeri olan *JavaBean* kavramından bahsetmemiz gerekir.

Kabaca bir tanım ile *JavaBean*, "Java'nın yeniden kullanılabilen, araçlar yardımı ile görsel olarak yönetilebilen yazılım bileşeni" olarak bilinir. Ancak bu tanımın gerçekten birşey ifade edebilmesi için "bileşen" kavramını bir parça daha açmalıyız.

Bileşen sözcüğü çok farklı bağlamlarda, "bir bütünü oluşturan parçalardan birine verilen ad" anlamına gelecek şekilde kullanılmaktadır. Yazılım dünyasında ise birçok farklı öge bileşen olarak adlandırılmaktadır. Örneğin İşlevsel Programlamada (functional programming) her bir işlev programın bütününü oluşturan bir bileşen gibi görülebilirken, benzer işlevsellikleri nedeniyle bir araya getirilmiş kod kesimlerinin oluşturduğu "modül"ler de bileşen olarak adlandırılabilmektedir.

Son yıllarda "bileşen (*component*)" sözcüğü yazılım bağlamında daha açık bir anlam kazanmış ve Bileşen Tabanlı Yazılım Mühendisliği (*Component-Based Software Engineering*) ya da Bileşen Tabanlı Yazılım Geliştirme (*Component-Based Software Development*) yaklaşımı çerçevesinde büyük önem kazanmıştır. Bileşen Tabanlı Yazılım Geliştirme yeni bir yaklaşım değildir ve yeni yazılımların oluşturulması için daha önceden geliştirilmiş yazılım bileşenlerinin bir araya getirilmesi fikrine dayanır. Ancak uzun yıllardır var olan bu yaklaşım, "yeniden kullanılabilir yazılım bileşenleri pazarı"nın oluşmasıyla birlikte yeniden önem kazanmıştır.

Bileşen Tabanlı Yazılım Geliştirme, herhangi bir programcının daha önce geliştirdiği kod parçalarını (işlevleri, sınıfları) bir araya getirerek yeni bir programda kullanmasından çok; kaynak kod düzeyine inilmeden, derlenmiş kod kesimlerinin bir takım biçimlendirme (*configuration*) araçları yardımı ile bir araya getirilerek yeni bir yazılım oluşturulması şeklinde gerçekleştirilir. Yazılımın geliştirilmesinde kullanılan bu "derlenmiş ve davranışları biçimlendirme araçları yardımıyla değiştirilebilen" ögelere "bileşen" adı verilir.

Bu yaklaşımda amaç; yazılımı oluşturmak için olabildiğince hazır bileşenleri kullanarak yazılım geliştirme maliyetlerini düşürmektir. Hazır bileşenlerin satın alınması, bu bileşenler için harcanacak olan geliştirme ve bakım maliyetlerinden kurtulmak anlamına gelir. Çünkü bu maliyet bileşeni satan kuruluş tarafından karşılanacaktır. Böylece "..geliştirme! Satın al!.." yaklaşımı ile yazılım gereksinimleri hem daha hızlı bir şekilde karşılanabilecek hem de kendi alanlarında uzmanlaşmış kuruluşlarca geliştirilmiş bileşenler kullanılacağından daha profesyonelleşmiş ürünler ortaya çıkacaktır. Tabi bütün bunların gerçekleşebilmesi için öncelikle "bileşen pazarının" gelişmesi, yazılım geliştiricilerin gereksinim duydukları bileşenlere ulaşabilmeleri gerekmektedir.

Günümüzde özellikle Web uygulamaları ya da dağıtılmış uygulamalar geliştirmek için kullanılan teknolojiler çoğunlukla bu tanıma uyan (derlenmiş ve davranışları biçimlendirme araçlarıyla değiştirilebilen)

bileşenlere dayanmaktadır. Bu bileşenler çoğunlukla "Uygulama Çatısı (*framework*)" adı ile karşımıza çıkmakta, bazı uygulama çatıları bir takım güçlü geliştirme araçları ile desteklenirken (.NET için Visual Studio ya da çeşitli Java Uygulama Çatıları için Eclipse eklentileri gibi), diğerleri için araç desteği daha sınırlı kalmaktadır. Öte yandan bunların bazıları ücretli iken birçok ücretsiz ve kaliteli ürün de bulunmaktadır. Başka bir deyişle "bileşen pazarı" hızla büyümektedir.

5.10.1 JavaBean Yazma Kuralları

JavaBean, Java'nın bileşen modelidir. Bu açıdan bakıldığında, her Java sınıfı bileşen değildir. Bir Java sınıfının *JavaBean* olabilmesi için aşağıdaki kurallara uyması gerekir:

1- Sınıfın varsayılan kurucusu olmalıdır

2- Sınıfın nitelikleri için erişim yöntemleri tanımlı olmalı ve bu yöntemler daha önce anlatılan isimlendirme kurallarına uymalıdır. Böylece derlenmiş durumdaki bir *JavaBean* nesnesinin içeriğine kodla ya da bir takım araçlar yardımı ile erişmek olanaklı olabilir.

3- Sınıf serileştirilebilir (*serializable*) olmalıdır. Böylece nesnenin durumu platformdan bağımsız olarak yazılabilir, okunabilir ya da ağ üzerinden aktarılabilir.

> ***Bir Java sınıfının bu kurallara uyması, o sınıfın JavaBean olarak adlandırılması için yeterli olsa da gerçek anlamda bir JavaBean, niteliklerine erişim olanağından daha fazlasını sunmalıdır. Bir JavaBean, kendisi ile ilgili olayların (events) dışarıdaki kod kesimlerinden algılanabilmesi, durumunun saklanabilmesi ya da davranış ve görünüşünün dışarıdan değiştirilebilmesi gibi amaçlara hizmet edecek kod kesimleri de içerecektir. Ancak bu kitabın içeriği ve hedef kitlesi açısından bu bölümde bu konulara girilmeyecektir.***

5.10.2 Serileştirme (*Serialization*)

İlk iki kuralı şimdiye kadar açıkladık. Şimdi serileştirmenin ne olduğunu ve nasıl yapılacağını açıklayalım.

Serileştirme, bellekteki bir nesnenin bütün alt nesne ağacıyla birlikte bir "*byte* serisi (*byte sequence*)" haline getirilmesidir. Oluşan veri bir dosyaya yazılabilir ya da ağ üzerinden başka bir makinadaki Java Sanal Makinasına aktarılabilir. Dosyadan okunan ya da ağ üzerinden alınan serileştirilmiş veri, bellekte yeniden Java nesnesine dönüştürülebilir (*deserialization*). Yeni oluşturulan nesne, orijinal nesnenin serileştirilmeden önceki halinin aynısıdır. Burada bir önemli nokta, serileştirilerek ağ üzerinden taşınan Java nesnesinin farklı bir platformdaki bir Java Sanal Makinasının belleğinde herhangi bir ek işleme gerek olmaksızın yeniden nesne haline getirilebiliyor olmasıdır.

Peki bir nesnenin serileştirilebilir olmasının faydası nedir? Bu noktada serileştirmenin kullanımına dair iki örnek verelim.

İlk örneğimiz, serileştirme yardımıyla etkin bellek kullanımıdır. Bellekteki her nesne kullanılabilir bellek alanından birazını işgal eder. Ancak herhangi bir anda bellekte bulunan birçok nesneye, aslında o anda ve belki yakın zaman diliminde erişilmeyecektir. Bu nesneler henüz Çöp Toplayıcı (*Garbage Collector*) tarafından da bellekten atılabilecek durumda olmayan, yani kendilerine referans olduğu için henüz çöp haline gelmemiş nesnelerdir. İşte bu noktada belleğin daha etkin kullanılabilmesi için izlenen bir yol, çalışma zamanında belirli bir süre erişilmeyen bazı nesneleri serileştirip diske yazarak o nesnenin bellekte işgal ettiği yeri boşaltmaktır. Aynı nesneye sonradan tekrar erişildiğinde, diskteki serileştirilmiş veri okunarak o nesne bellekte yeniden oluşturulur ve hizmet vermesi sağlanır. Böylece bellek daha etkin kullanılmış olur. Bu teknik, uygulama sunucuları gibi başarımın çok önemli olduğu büyük ve karmaşık sistemlerde uygulanmaktadır.

İkinci örneğimiz ise serileştirmenin bir başka kullanım alanı olan Uzaktan Yöntem Çağırma (*Remote Method Invocation – RMI*) olacaktır. Uzaktan Yöntem Çağırma, bir programı oluşturan nesnelerin çeşitli (başarımı yükseltmek, güvenlik, yedekleyerek çalışma.. gibi) nedenlerle farklı Java Sanal Makinaları üzerinde çalıştırılmasına rağmen, bu nesnelerin kullanıcı

tarafından tek bir sanal makinada çalışıyormuş gibi görünmesini sağlar. İşte bu noktada; farklı sanal makinalarda çalışmakta olan nesnelerin birbirlerinin yöntemlerini çağırırken parametre göndermesi ya da yöntemden geriye değer döndürmesi söz konusu olduğunda serileştirme devreye girer. Çünkü iki farklı makina arasında aktarılacak olan veri aslında bir Java nesnesidir ve çalışma zamanında oluşturulmuştur. Bu nesnenin bir A makinasından B makinasına aktarılması için ağ altyapısından faydalanılacak olsa da, A makinasındaki nesnenin B makinasında yeniden oluşturulabilmesi gereksinimi serileştirme yardımı ile karşılanır. Gönderen taraf nesneyi serileştirir, ağ üzerinden diğer makinaya aktarır, serileştirilmiş nesneyi alan taraf bunu bellekte yeni bir nesne oluşturmak için kullanır ve program böylece çalışıp gider.

Serileştirmenin bu gibi kullanım alanları, daha çok Kurumsal (*Enterprise*) ölçekteki konularla ilgilenen *Java Enterprise Edition (JEE)* bağlamında kaldığından daha fazla ayrıntıya girmeyecek, hemen bu noktada bir sınıfın serileştirilebilmesi için ne yapılması gerektiğini açıklayacağız.

5.10.3 `java.io.Serializable` Arayüzü

Bir Java sınıfının serileştirilebilir olduğu, `java.io.Serializable` arayüzünü gerçekleştirmesi yoluyla bildirilir. Bu arayüz bir işaretçi arayüzdür (*marker* ya da *tagging interface*). İşaretçi arayüz, herhangi bir yöntem içermeyen arayüz anlamına gelir. `java.io.Serializable` arayüzü de herhangi bir yöntem içermeyen, ancak derleyiciye bu arayüzü gerçekleştiren sınıfın serileştirilebilir olup olmadığını denetlemesini söyleyen bir arayüzdür. Derleyicinin bir sınıfın serileştirilebilir olup olmadığını denetlerken kullanacağı bazı kurallar vardır:

1- Temel türler serileştirilebilirdir.

2- Serileştirilemeyen nitelikler içermeyen sınıflar serileştirilebilirdir.

3- Bir sınıf serileştirilemeyen bir nitelik içeriyorsa ve bu sınıfın yine de serileştirilebilir olması isteniyorsa, o nitelik `transient` anahtar

sözcüğü ile işaretlenerek, bu sınıfa ait bir nesne serileştirilirken bu niteliğin serileştirilmeyeceği belirtilir.